

Moderne Programmiermethoden

Eine Ideensammlung

**Diplomarbeit zur Erlangung des akademischen Grades Diplominformatiker /
Diplomingenieur vorgelegt der Fakultät für Informatik und Automatisierung der
Technischen Universität Ilmenau von**

Robert Will

Verantwortlicher Hochschullehrer:	Univ.-Prof. Dr.-Ing. habil. D. Reschke
Hochschulbetreuer:	Dr.-Ing. Günter Hübel
Tag der Ausgabe des Themas:	20. Oktober 2003
Tag der Abgabe der Diplomarbeit:	20. Juli 2004
Matrikelnummer:	29335
Inventarisierungsnummer:	2004-04-20/042/IN99/2251

Selbstständigkeitserklärung

Ich erkläre hiermit, dass

1. ich alle verwendeten Publikationen vollständig deklariert habe,
2. ich alle helfenden Personen in Abschnitt 8.3 genannt habe,
3. alle anderen Ideen, Materialien und Formulierungen ausschließlich von mir stammen,
4. und dass alle Werturteile und subjektiven Einschätzungen meine persönlichen sind und nicht die meines Betreuers oder der Institution.

(Robert Will)

Inhaltsverzeichnis

1	Einführung	1
1.1	Die Herausforderung	1
1.2	Die neue Sicht: ein Überblick	1
1.2.1	Korrektheit versus Konsistenz	1
1.2.2	Gute und böse Redundanz	2
1.3	Mit Methode?	3
1.4	Inhalte dieser Arbeit	4
1.4.1	Grundlagen	4
1.4.2	Hauptteil	5
1.4.3	verwendete Fallstudien	5
1.5	Eine Ideensammlung	6
2	Qualität	7
2.1	Äußere Qualität	7
2.2	Grenzeigenschaften: Korrektheit und Effizienz	8
2.3	Innere Qualität	9
2.4	Grenzen der Unabhängigkeit	10
2.5	Grenzen der Wiederverwendbarkeit	11
3	Formale Beschreibungen	13
3.1	Qualität von Beschreibungen, insbesondere Objektivität	13
3.2	Elementare Beobachtungen und Begriffsbestimmungen	14
3.3	Eine kleine Phänomenologie	16
3.4	Fallstudie Eins: Mitarbeiterverfügbarkeit, logisch	17
3.5	Einschränkungen und Kontrolle	18
3.6	Fallstudie Zwei: Diplomacy	20
3.6.1	Die neue Beschreibung	21
4	Formale Theorien der Programmierung	25
4.1	Alte und neue Theorien	25
4.1.1	Operationale Semantik	25
4.1.2	Axiomatische Semantik	26
4.1.3	Denotationale Semantik	27
4.1.4	Algebraische Semantik	29
4.1.5	Prädikative Semantik	31
4.2	Eine Praktische Theorie der Programmierung	32
4.2.1	Algebra	33
4.2.2	Rekursive Verfeinerung	33
4.2.3	Laufzeitschranken	34
4.2.4	Routinenaufruf	35
4.2.5	Schlussfolgerungen	35

5	Ansätze einer allgemeinen Theorie	36
5.1	Versprechen und Grenzen des Formalen Ansatzes	36
5.1.1	Voll-Formal versus Halb-Formal	37
5.1.2	Methode informal?	38
5.1.3	Herausforderungen einer modernen Programmiermethode	38
5.2	Der Dreibund	40
5.2.1	Was sind Spezifikationen?	40
5.2.2	Spezifikationen versus Tests	41
5.2.3	Testen von Spezifikationen	42
5.2.4	Ziele von und Vorgehen bei Tests	43
5.3	Die Operationalisierungshierarchie	43
5.3.1	Effizienz als Grundanliegen, Optimierung als Beiwerk	45
6	Programmbausteine	47
6.1	Ausdrücke und Funktionen	48
6.2	Einfache Seiteneffekte	49
6.2.1	Ein Schleifenidiom	50
6.2.2	Prozeduren	51
6.2.3	Routinierte Abstraktion	52
6.2.4	Die Vorteile von Vorbedingungen	55
6.3	Fallstudie: Schrittweise Verbesserung einer großen Routine	56
6.4	Abstrakte Datentypen	61
6.4.1	Module	61
6.4.2	Prozeduren versus Funktionen	65
6.4.3	Zum Beispiel Iteratoren	67
6.4.4	Abstrakte und andere Datentypen	69
6.5	Fallstudie: flexible Arrays	71
6.6	Zu erben ist besser als zu kaufen	75
6.6.1	Vererbung als Verfeinerung	75
6.6.2	Vererbung und Dynamische Bindung	77
6.6.3	Vererbung und Statische Bindung	79
7	Entwurf durch Alternativen	82
7.1	Entwurf formal gesehen	82
7.2	Entwurf als Prozess	84
7.2.1	Pragmatik von Schnittstellen	85
7.2.2	Alternativen	86
7.3	Alternativen in OO	87
7.3.1	Variablen, Funktionen, Ausdrücke	88
7.3.2	Prozeduren	91
7.3.3	Wohin mit der Routine?	91
7.3.4	Unterarten oder Unterklassen?	92
7.3.5	Werte, Objekte oder Zustände?	94
7.3.6	Vererbung oder Delegation?	98
7.3.7	Singleton oder Multiton?	99
7.3.8	Refaktorisieren abstrakt	99
8	Und weiter?	102
8.1	Was ich erreicht habe	102
8.2	Was man mal erreichen sollte	104
8.3	Danksagungen	104
A	Thesen	108

1 Einführung

1.1 Die Herausforderung

“Programmierung ist die (formale) Herleitung von Programmen aus (ebenso formalen) Spezifikationen.”

— Edsger W. Dijkstra

“Der Programmcode ist die Spezifikation.”

— Aus dem Manifest der extremen Programmierung.

“Softwareentwicklung ist die Umwandlung menschlichen Verlangens in ausführbare Spezifikationen.”

— Eine einheitliche Sicht?

In diesen Zitaten treffen zwei völlig entgegengesetzte Ansichten aufeinander. Die eine sieht die Programmierung als eine rein mathematische Angelegenheit an, die andere eher als eine Kunstfertigkeit, deren formal-mathematische Aspekte vernachlässigbar sind. Formalisten arbeiten mit mathematischen Minimal-Programmiersprachen und vielen logischen Formeln, während die Programmier-Artisten auf die Beherrschung aller Einzelheiten ihrer objektorientierten (OO) Programmiersprachen und -werkzeuge Wert legen.

Die formalen Methoden bestehen mit Ihrer Eleganz: man kann Programmierprobleme ganz systematisch durch Rechnen lösen, man muss dabei gar nicht so viel überlegen, man kann es lernen. Aber leider scheint die Praxis nicht die passenden Rechenaufgaben zu bieten.

Diese Arbeit behandelt die Programmierung, oder etwas allgemeiner, die *Konstruktion* von Programmen. Wir beschäftigen uns also nicht mit der vorangehenden Analyse, nicht mit der Auswahl von Komponenten, und auch nicht mit dem Management. Unsere Prämisse ist: Es ist Programmcode zu schreiben — wie macht man das am Besten?

1.2 Die neue Sicht: ein Überblick

Softwareentwicklung dient der *Automatisierung* monotoner intellektueller Prozesse. Dies bedingt eine *Präzisierung* der selbigen. Diese Präzisierung findet zunächst auf einem zwischenmenschlichen Niveau statt: alle Beteiligten (Menschen mit ganz verschiedenen Denkweisen und Wissensbereichen) müssen sich über die Prozesse einig sein. Dann muss man die Prozesse *formalisieren*, also in eine mathematisch eindeutige Sprache bringen, und schließlich muss man sie *operationalisieren* also in eine durch Computer ausführbare Sprache bringen. Diese Schritte des Formalisierens und Operationalisierens sind für Laien nicht leicht zu unterscheiden; das ist eine Spezialität von Informatikern! Wahrscheinlich muss es auch eine Spezialität von Informatikern sein, dass zwischenmenschliche Präzisieren zu fördern. In dieser Arbeit behandeln wir aber nur die Programmierung, worunter wir verstehen: formalisieren und operationalisieren. Die obigen Zitate betrachten einerseits das Formalisieren und Operationalisieren als einen einzigen ganzheitlichen Schritt (Extreme Programmierung), beziehungsweise andererseits den Schritt vom Formalisierten zum Operationalisierten als wichtigsten Schritt, den man getrennt vom Rest durchführen kann (formaler Ansatz).

1.2.1 Korrektheit versus Konsistenz

Formalisten kümmern sich nur um die formale *Korrektheit* ihrer Programme. Diese Sicht setzt aber eine formale Spezifikation voraus. Die Artisten auf der anderen Seite wollen ihr Programm

nur an informalen Anforderungen und Geschäftsmodellen messen — Gebiete, die für Beweise nicht zugänglich sind. Oft lassen sich aber gerade die Fehler in den Programmen nicht in diesen Begriffen ausdrücken: *NullPointerException*, *ArrayIndexOutOfBounds*, ... — das sind alles formal behandelbare Probleme. Wenn man also mangels formalen Anwendungswissen schon nicht die Korrektheit eines Programms beweisen kann, so sollte man doch wenigstens die Abwesenheit von solch offensichtlich unerwünschten Ereignissen formal absichern können!

Unsere Lösung ist, das absolute Kriterium der Korrektheit durch ein Kriterium der Konsistenz zu ersetzen. An Stelle uns von einer Spezifikation abhängig zu machen, die zu 100% mit dem Programmcode redundant ist, streben wir nach einer *Konsistenz* von Spezifikationsfragmenten, ausführbarem Code, Zusicherungen und Tests. In dieser Aufzählung ist nur der ausführbare Code unabdingbar — all die anderen Komponenten kann man je nach Bedarf hinzufügen, in der jeweils notwendigen Menge.

Ein entscheidender Punkt ist, dass man bei jedem nicht-trivialen Problem eigentlich immer alle diese Arten der Redundanz braucht, dass aber andererseits jedes nicht-triviale Programm sehr viele triviale Bestandteile enthält und diese könnte man nicht *produktiv* entwickeln, wollte man sie alle komplett unabhängig spezifizieren. Ich kann hier schon vorweg nehmen, dass das richtige Maß dieser Redundanz ein Erfahrungswert ist, der von vielen Faktoren abhängt. Diese Arbeit beschränkt sich darauf, die Möglichkeiten der “Redundanzierung” aufzuzeigen und überlässt es dem Entwickler sich daraus zu bedienen. Ohnehin entwickelt sich ein gutes Maß von Redundanz während der Programmierung, wenn man zum Beispiel problematischen Code mit Zusicherungen spickt und mit Testfällen stichelt (zum Beispiel um einen Fehler ausfindig zu machen, oder sich nach dessen Beseitigung der Abwesenheit weiterer Fehler zu versichern), oder wenn man eine Spezifikation wieder entfernt, weil sie dem Code zu ähnlich geworden ist.

An Stelle von formaler Korrektheit, sprechen wir nur noch von der *Konsistenz* eines Programms. Unser Ziel ist es, komplett konsistenten Code zu schreiben.

1.2.2 Gute und böse Redundanz

Extreme Programmierer (und andere Verfechter moderner Programmiersprachen) legen viel Gewicht auf die Vermeidung von Redundanz im Code, damit dieser stets leicht geändert werden kann; Code soll beweglich (agil) bleiben, dann kann man auch Fehler leichter beseitigen. Fast alle programmiersprachlichen Konstrukte (und Entwurfsmuster) dienen diesem Prinzip, insbesondere die Vererbung! Auf der anderen Seite erfordert jede Art von Beweis Redundanz: die Schritte eines Beweises sollen “ganz klar” aufeinander folgen, aber alles, was “ganz klar” folgt, braucht man eigentlich nicht extra zu sagen, geschweige denn aufzuschreiben. Sind die Methoden also inhärent widersprüchlich?

Nehmen wir einmal an, höhere Programmiersprachen sind gegen eine Art von “böser Redundanz”, aber formale Methoden arbeiten mit einer Art von “guter Redundanz”. Wodurch können wir die beiden unterscheiden? Zunächst erstmal rein gefühlsmäßig beim Lesen von Code: böse Redundanz stört den Leser und verwirrt ihn: “hatten wir das nicht dort schon einmal?” denkt er sich, oder “das ist doch nochmal genau dasselbe!” Gute Redundanz dagegen hilft dem Leser: “Ach deswegen ist das so!” Oder nach der Lektüre einer Spezifikation: “Alles klar, dann brauche ich das [nämlich die Implementierung] ja nicht mehr lesen.” In der Tat wird so ein Ansatz von Harlan Mills (dem Erfinder des “Cleanroom Software Engineering”) verwendet[36]: aller Code wird durch Lesen verifiziert und die Annotationen dienen dazu den kritischen Leser von der Korrektheit des Codes zu überzeugen. Das erfordert natürlich sehr viel Disziplin! Die moderne Variante bedient sich des Computers zum Überprüfen der Annotationen, ganz genauso wie man sich des Computers bedient, um böse Redundanzen zu beseitigen! (Compiler generieren Maschinencode aus Code höheren Niveaus.)

Gute Redundanz ist also nicht simple Wiederholung, sondern sie bietet *verschiedene Sichten* auf ein und dasselbe Objekt. Nach dem klassischen Schema ist eines die Anwendersicht

(Spezifikation) und das andere die Implementierungssicht (Programm); dies lässt sich aber verallgemeinern: man kann zum Beispiel mehrere Sichten auf ein schwieriges Problem haben, oder mehrere Verfeinerungsstufen einer Routine (dazu kommen wir noch), oder einfach eine “Hilfsansicht” inmitten von schwierigem Code (das sind zum Beispiel Schleifeninvarianten). Im besten Fall ist nun die Konsistenz all dieser Sichten der guten Redundanz automatisch überprüfbar: entweder zur Compile-Zeit (so wie Typen) oder zur Laufzeit des Programms (so wie Zusicherungen). Extreme objektorientierte Methoden empfehlen, möglichst alle Annotationen mit Hilfe des Typ- und Klassensystems auszudrücken. Das hat natürlich durch die statische Prüfung prinzipielle Vorteile, aber man muss sich bewusst sein, dass es nicht immer möglich ist. (Von einem Compiler, der alle Annotationen statisch überprüfen kann, wäre es nicht mehr weit zu einem Compiler, der direkt Spezifikationen in (effizient) ausführbare Programme übersetzt. Dann gäbe es nichts mehr zu prüfen.) Man kommt also nicht darum herum, auch die Laufzeitüberprüfungen zu nutzen.

Und das soll die große Perspektive für die Zukunft sein:

Mit höheren Programmiersprachen und gutem Entwurf vermeiden wir so viel *böse Redundanz*, dass Programmcode auf einem hohen Niveau anwendungsnah geschrieben werden kann. Durch Unterstützung formaler Methoden fügen wir so viel *gute Redundanz* ein, dass alle technischen Fehler ausgeschlossen werden.

Die Theorie kommt der Praxis entgegen durch die *Verfeinerung*: Wir machen Schluss mit der strengen Dualität von Spezifikation und Programmen; stattdessen kann man beides in der selben Sprache ausdrücken und man ist nicht gezwungen selbst-erklärende Programme zu spezifizieren oder bereits ausführbare Spezifikationen zu implementieren. Die Praxis kommt der Theorie entgegen, indem sie all’ ihre Sprachkonstrukte im theoretischen Rahmen bewertet und nur diejenigen behält, die in der Methode einen sinnvollen Platz haben. Der Vorteil ist, dass wir für die verbleibenden Konstrukte eine klare Einsatz-Methode haben, und die anderen Konstrukte brauchen wir einfach nicht mehr.

Bertrand Meyers EIFFEL Methode[29] verfolgt bereits den gleichen Ansatz. Wir bauen darauf auf, indem wir die formalen Konzepte noch mehr herausarbeiten. Gerade dies wird durch die modernere formale Sicht der Verfeinerung unterstützt.

Der Leser hat schon bemerkt, dass ich noch zwischen der Terminologie gute/böse bzw. positive/negative Redundanz schwanke. Letzteres hat den Vorteil wissenschaftlicher zu klingen, aber damit verspricht es vielleicht mehr als es hält. Die gut/böse Variante macht hingegen einerseits die Wichtigkeit der Unterscheidung klar, andererseits aber auch die Ungenauigkeit der Begriffe.

Ich zitiere hier Werke von Mills und Meyer, aber nichts zur Extremen Programmierung (XP). Von dieser verarbeite ich hier nämlich nur die Grundprinzipien, wie sie auf unzähligen privaten und interaktiven Webseiten (so genannten WiKis) und in sekundären Publikationen zu finden sind. Nur der Vollständigkeit halber nenne ich eine Person, die als Leitfigur der Bewegung angesehen wird: Kent Beck. Überhaupt ist XP so eine Art “Basisbewegung” von Programmierern für Programmierer, und wissenschaftliche Arbeiten untersuchen das Phänomen nur a posteriori und prägen es überhaupt nicht mit.

1.3 Mit Methode?

Die Aufgaben einer Methode sind:

1. Dem Entwickler den gesamten Raum der Möglichkeiten klar zu machen, mit allen dunklen Ecken und allen Querverbindungen.
2. Diesen Raum soweit wie möglich einzuschränken, damit der Programmierer weniger Entscheidungen zu treffen hat.

Zu einem tiefen Verständnis des Lösungsraumes gehört es, nicht nur seine verschiedenen Bestandteile zu kennen, sondern auch deren Zusammenhänge genau zu verstehen. Bei einer Programmiersprache heißt das vor allem zu verstehen, welche Konstrukte ähnlichen Zwecken dienen, und worin genau deren Unterschiede bestehen. Es ist ja ganz klar, dass selbst eine “minimale Programmiersprache”, wie ich sie im Folgenden empfehle, weitaus mehr Möglichkeiten enthält als zur Turing-Vollständigkeit nötig wäre. Es gibt also notwendigerweise Redundanzen und diese muss man sich bewusst machen, um je nach Anforderung die passenden Konstrukte auszuwählen. Viel zu oft treffen Programmierer die Auswahl noch nach Mustern, nach dem Motto “das habe ich doch so ähnlich schon mal gesehen”, und dabei führen sie Teile einer anderen Lösung in ihr Programm ein, die vielleicht überhaupt gar nicht gebraucht werden. Natürlich ist Denken in Mustern und Schöpfen aus Erfahrung nicht schlecht. Wenn man vorhandene Erfahrungen aber auf ein neues Problem anwenden will, so muss man sie auf ihre Elementarbestandteile reduzieren und nur das Nötige auswählen. In der Programmierung soll es ja um das *Kombinieren* von Konzepten gehen, und nicht das ständige Hinzufügen neuer Konzepte oder Muster.

Wenn wir nun den Lösungsraum bis ins kleinste Detail verstanden haben, warum wollen wir dann, dass ihn die Methode wieder möglichst radikal einschränkt? Antwort: weil der Lösungsraum einfach zu groß ist. Und diese Größe kommt ja nicht durch sein völliges Ausleuchten zu Stande. Diese Größe ist schon da, und wir machen sie uns nur bewusst. Den Lösungsraum einschränken bedeutet Entscheidungen vorweg nehmen, die man sonst jedes Mal neu treffen müsste! Mit jeder solchen Entscheidung, die uns die Methode vorgibt, sparen wir ein bisschen Arbeit und gewinnen eine gewisse Sicherheit, dass die Entscheidung auch richtig getroffen wurde. Wenn wir natürlich verlangen, dass jede unserer Lösungen genau auf das Problem passt, ist es besonders schwierig für eine Methode Entscheidungen a priori vorwegzunehmen. Deswegen gibt es ja so wenig gescheite Methoden!

Man muss dabei bedenken, dass das Einschränken des Lösungsraumes nicht nur die besseren Lösungen auswählen und schlechtere ausblenden soll — oft kann man ja gar nicht sagen, welche Lösung nun besser ist, und schon gar nicht im Vorhinein. Beim Einschränken des Lösungsraumes kann man radikaler vorgehen und alle Lösungen verwerfen, die nicht viel besser sein können, als mindestens eine, die man nicht verwirft! Dieses radikale Vorgehen hat den Vorteil, dass man Entscheidungen schnell und systematisch trifft, und dass die resultierende Lösung so weit wie möglich standardisiert ist. Dabei ist natürlich immer die Balance zu halten, zwischen einer Lösung, die sich nicht mehr dem Problem anpasst, und einer Lösung, die zu extravagant ist.

Viele Methoden gehen in beiden Richtungen nicht weit genug, weder bei der Erforschung des Lösungsraumes, noch bei dessen Einschränkung. Das Wort “Methode” ist zusammen mit der Objektorientierung in großen Umlauf gekommen, aber leider hat es da eine besonders schwache Bedeutung: die OO-Methoden geben meist nur eine bestimmte Menge von Diagrammen vor, die man in einer bestimmten Reihenfolge erstellen soll, und dann wird alles gut. Leider sind diese Diagramme aber meist viel zu weit von der Aufgabe entfernt, und vom Entwurf eines Programms auch. In dieser Arbeit konzentrieren wir uns auf Inhalte, nicht auf die Form.

Siehe zu diesem Thema auch das Kapitel über “Methoden” in [32].

1.4 Inhalte dieser Arbeit

1.4.1 Grundlagen

Qualität: Bevor man etwas anfängt, sollte man erstmal wissen, was man eigentlich will. Ganz klar. Und die Softwareentwicklung will ganz klar *Qualität* — aber nicht mehr als man auch bezahlen kann. Also wollen wir auch *Produktivität*. Leider gibt es aber über die Bedeutung (und vor allem: den Umfang) dieser Begriffe keinen Konsens. Oft werden wichtige Qualitätsmerkmale außer Acht gelassen, und man meint, sie dann nachträglich noch einbauen zu müssen. (“Das Programm ist super, aber wie machen wir es jetzt noch robust?”) Auch kommt es vor, dass manche Methoden Aspekte behandeln, die eigentlich gar nicht in die Programmierung gehören, sondern in eine frühere Phase oder ins Projektmanagement, oder woanders hin. Indem wir Qualitäts-

merkmale von Software kategorisieren, filtern wir genau jene heraus, die für die Programmierung wichtig sind. Letztens kommt es auch vor, dass Methoden Anforderungen an die Programmierung stellen, die sie gar nicht erfüllen kann. Hier müssen wir natürlich Einhalt gebieten und realistische Anforderungen stellen.

Formale Theorien der Programmierung: Wenn man sich über diese Theorien informieren will, dann kann man natürlich ein Buch darüber lesen. Warum führe ich sie hier also trotzdem auf? Es geht darum, zu zeigen, welche dieser Theorien für die Praxis der Programmierung nützlich sind, worin dieser Nutzen besteht und wie man diesen Nutzen erzielt. Außerdem sagen die Theorien selbst nicht explizit worüber sie überhaupt Aussagen machen: welche Phänomene betreffen sie? Nur [21] geht darauf explizit ein, und [16] macht es sich zu Nutze, um die Theorie möglichst einfach und praktisch zu machen. Zu guter Letzt gibt es auch Theorien, die heutzutage schlicht veraltet sind. Natürlich sind diese Theorien nicht plötzlich falsch, aber oft ist die neue Theorie einfach nützlicher, oder allgemeiner, oder besser auf die Programmierung zugeschnitten. Alte Theorien basieren eben noch auf der starken Trennung von Programm und Spezifikation, und das ist uns heutzutage schlicht im Weg.

Bedeutung formaler Beschreibungen: Aus der Beschäftigung des Informatikers mit formalen Dokumenten (wie zum Beispiel seinen Programmen), insbesondere wenn Beweise im Spiel sind, folgt natürlich, dass man sich über deren Bedeutung absolut im Klaren sein muss. Ein Beweis ist eben nur so gut wie seine Voraussetzungen. Dies gilt nicht nur für die Verwendung formaler Methoden, sondern hilft generell bei der Erstellung präziser Dokumente (die dabei auch nicht zu lang geraten und gut strukturiert sein sollen). Ein Ausflug in die angewandte Erkenntnisphilosophie.

1.4.2 Hauptteil

Die neue Theorie über das Wesen der Programmierung habe ich ja oben schon kurz skizziert. In diesem Kapitel stürzen wir uns ganz hinein, untersuchen den Begriff der Spezifikation genauer, seinen Zusammenhang zu Tests, und lernen die Operationalisierungshierarchie kennen.

In *Bausteine der Programmierung* legen wir den Lösungsraum offen, von seinem einfachsten Bestandteil bis zum zweitschwersten. Gleich einer Einführung in die Programmierung durchfliegen wir den Raum der wichtigsten Konstrukte einer Programmiersprache. objektorientierte und funktionale Programmierung werden dabei im selben Rahmen betrachtet, und Spezifikationen und Code gehen stets Hand in Hand. Besonders wichtig ist die *Vererbung* am Ende des Kapitels.

Ein Sprachkonstrukt namens “Exceptions” erfordert besonders viel Aufmerksamkeit. Leider ist aber in dieser Arbeit nicht genug Platz, deswegen verweise ich hier nur auf zwei meiner vorherigen Veröffentlichungen zum Thema *Notfallbehandlung*: [48, 47].

Entwurf mit Alternativen: Im letzten Kapitel geht es um den Programm-Entwurf. Als Entwurf bezeichne ich eine Menge von Modul-Spezifikationen (bzw. Klassenspezifikationen) und die Methode besteht hauptsächlich darin, aus einer Reihe von alternativen Entwürfen den (für das gegebene Problem) besten Entwurf auszuwählen. Außerdem werden wir sehen, wie sich Methoden des Refactoring schon von Anfang an zum Guten einsetzen lassen.

1.4.3 verwendete Fallstudien

In Abschnitt 3.6 spezifiziere ich die Spielzug-Auswertung des Strategiespieles Diplomacy, zuvor von mir veröffentlicht in [49].

In Abschnit 6.5 behandle ich die Invariante einer modernen Datenstruktur, die ebenfalls schon länger veröffentlicht ist [46].

Und in Abschnitt 6.3 verbessere ich eine Routine, die aus einem größeren Open-source-Programm stammt.

1.5 Eine Ideensammlung

Diese Arbeit ist eine Ideensammlung im positiven wie auch im negativen Sinn.

Ideensammlung im positiven Sinn: ich präsentiere nicht einfach ein paar meiner (vielen) Ideen, sondern ich wähle aus fünfzig Jahren Forschung und Praxis des Programmierens diese Ideen, die bis heute durchgehalten haben, Prinzipien mit großer Allgemeingültigkeit, und deren moderne Fortsetzung (im Gegensatz zum letzten Schrei).

Ideensammlung aber auch im negativen Sinn: ich zeige das Zusammenspiel dieser Ideen auf: immer wieder muss man verschiedene Sichten einnehmen und ein wesentlicher Beitrag dieser Arbeit ist es eine Gesamt-Sicht aufzubauen, in der sich die einzelnen Ideen nicht widersprechen. Und hier kommt das Negative zum Vorschein: die Gesamtsicht enthält nämlich genauso viele Lücken wie sie Ideen vereinigt. Damit muss ein Anwender der Methode eben leben. Und die Forschung hat hier noch kräftig zu tun! Zumindest gilt aber: das "Material zwischen den Lücken", das ich hier präsentiere, hat unter verschiedenen Sichten Bestand, man kann sich darauf verlassen.

2 Qualität

Zunächst erstmal unterscheiden wir Prozessqualität und Produktqualität, aber wie schon in der Einleitung angedeutet, behandeln wir hier nur Produkte und keine Prozesse. All die guten Eigenschaften, die ein Produkt haben soll, fasst man unter dem Begriff der Qualität zusammen. Die Qualität ist also per Definition die oberste und einzige Eigenschaft (Merkmal) eines guten Programms. Bei Software unterscheidet man zwischen inneren und äußeren Qualitätsmerkmalen (QM). Die äußeren QM sind alle Eigenschaften, die den Einsatz der Software betreffen: Funktionalität, Benutzerfreundlichkeit, Sicherheit, Robustheit, Effizienz. Die inneren QM sind diejenigen, die Wartung und Weiterentwicklung betreffen: Korrigierbarkeit (Fehler beseitigen), Portabilität (technische Komponente ändern, z.B. Datenbank, Betriebssystem), Änderbarkeit (fachliche Änderung) und Wiederverwendbarkeit. Wenn man eine *Software als Beschreibung* eines bestimmten Verhaltens ansieht, dann beziehen sich die äußeren QM auf Eigenschaften dieses Verhaltens, und die inneren QM auf die Eigenschaften der *Beschreibung* des Verhaltens. Spezifikation und Implementierung einer Software beschreiben dasselbe Software-Verhalten, die Implementierung ist nebenbei noch eine ausführbare Beschreibung dieses Verhaltens. Die äußeren Qualitätsmerkmale einer Software lassen sich folglich schon alle an der Spezifikation ablesen. Sie müssen also auch nur in der Spezifikationsphase behandelt werden!

Alle Anforderungen an ein Produkt, sind Anforderungen an dessen Qualität (oder den Preis). Üblicherweise wird zwischen funktionalen und nicht-funktionalen Anforderungen unterschieden und in diesem Kapitel geht es auch darum, dass man bei dieser Unterscheidung vorsichtig sein soll. Eine Warnung kommt von Michael Jackson (nicht dem Sänger)[22, Seite 342]:

Most of these requirements are ‘non-functional’ only in the sense, that they have not been analysed — yet. [...] Look sceptically at any classification of a requirement as ‘non-functional’. If you allow requirements like these to be relegated to the ‘non-functional’ category, you risk missing a large part of your problem.

Jackson schiebt also dem *Analysten* die Pflicht zu, nicht-funktionale Anforderungen in funktionale umzuwandeln. Für uns als Programmierer heißt das dann: wir haben es fast nur noch mit funktionalen Anforderungen zu tun und Anforderungen an die Innere Qualität. Der folgende Abschnitt begründet Jacksons Behauptung (was der in seinem dicken Buch nämlich nicht tut).

2.1 Äußere Qualität

In der Anforderungsanalyse gibt man zuerst (Qualitäts-)Ziele an, und spezialisiert diese dann zu Anforderungen und der Spezifikation. Die Ziele dienen dazu (1) den Prozess in Gang zu setzen und ihm Grenzen zu setzen, (2) die Anforderungen zu strukturieren und Entscheidungen zu begründen, und (3) Konflikte möglichst auf hohem Niveau zu lösen. Die Spezifikation ist eine rein funktionale Beschreibung des Softwareverhaltens. Alle nicht-funktionalen Anforderungen und Ziele (siehe Qualitätsmerkmale oben) finden sich *indirekt* in der Spezifikation wieder, sie bestimmen die Details, die in den funktionalen Anforderungen und Zielen fehlen. So gesehen geben die funktionalen Ziele und Anforderungen an, *was* die Software machen soll, und die nicht-funktionalen Ziele und Anforderungen schränken das *Wie* ein.

Zum Beispiel soll eine spezielle Textverarbeitung dabei helfen, Briefe zu schreiben (Funktion), und sie soll es so tun, dass Benutzer leicht damit zurecht kommen, dass sie so wenig wie möglich eingeben müssen, und dass die Briefe möglichst gut ankommen

(nicht-funktionale Ziele). Das “gute Ankommen” der Briefe geht dabei absichtlich darüber hinaus, was die Software (allein) erreichen kann. Es ist ja hauptsächlich Sache der Post (“gut ankommen” im wörtlichen Sinn) bzw. des Schreibstils (“gut ankommen” im übertragenen Sinn). Trotzdem kann auch die Software zur Erreichung dieser Ziele *beitragen*.

Im Laufe der Anforderungsanalyse untersucht man, was diese Ziele genau bedeuten und woran man Zielerreichung erkennt; dann spezifiziert man, wie gut die einzelnen Ziele erreicht werden sollen und folgert dann die Konsequenzen für die Funktion der Software. Man braucht dazu Wissen über den Einsatzbereich der Software: erstens, um relevante Angelegenheiten und Besorgnisse (engl. *concerns*) zu kennen, und zweitens natürlich, um Lösungsmöglichkeiten zu kennen. Dieses Anwendungswissen (*domain knowledge*) ist auch nötig, um die Ziele überhaupt erst einmal zu verstehen und zu erkennen, wie man sie befördern kann (ich schreibe absichtlich nicht: erreichen).

Im Beispiel kann man das “gute Ankommen” physikalisch interpretieren: damit der Brief den Empfängerbriefkasten erreicht, sollte die Ziel-Adresse existieren, sie sollte zu der vom Autor gemeinten Person gehören, und sie sollte in einem Post-lesbaren Format auf dem Brief stehen. (Das Programm kann auch je nach Briefgröße an die entsprechende Frankierung erinnern.) Ziel-Adressen kann man in einem Adressbuch sammeln und mit dem Telephonbuch (CD, Internet) validieren; dass der Benutzer auch die “gemeinte” Adresse auswählt, erreicht man durch klare Schnittstellen und evtl. durch die Möglichkeit Adressbucheinträge zu kommentieren (Spitznamen, Funktionsbezeichnungen); das Post-lesbare Format letztlich kann einfach spezifiziert und vom Programm durchgesetzt werden — es ist hier allerdings eine Frage der Robustheit, was passiert, wenn jemand an eine Adresse ungewöhnlichen Formats schreiben möchte, vielleicht noch dazu an eine, die nicht im Telephonbuch steht (Ausland, Krachl-Alm, Internatszimmer).

Dies sollte zeigen, wie die Qualitätsziele schrittweise auf Programmfunktionalität und -verhalten abgebildet werden. Wie man es schafft, dabei nicht abzuschweifen und nur die notwendigen und wirklich nützlichen Funktionen zu spezifizieren, lehren in diesem Beispiel die Methoden des Usability Engineering, allen voran die Nutzertests.

Ich habe diesen Prozess nur mal skizziert, weil in vielen Programmierübungen Tätigkeiten besprochen werden, die eigentlich in die Anforderungsanalyse gehören. Andererseits gibt es eine gewisse Analogie der Anforderungsanalyse zur Programmierung: so wie die äußeren Qualitätsmerkmale Ziele zu Spezifikationen konkretisieren, wird uns die innere Qualität leiten, diese Spezifikationen zu implementieren. Eine ähnliche Analogie wird in [54] aufgezeigt, jene ist aber viel eingeschränkter und m.E. unrealistischer: Die Umwandlungen *Anforderung* → *Spezifikation* und *Spezifikation* → *Implementierung* werden jeweils als *logische Transformationen* gesehen. Qualität darf nur noch dort wirken, wo Indeterminismus besteht; und weil das Modell von völlig formalen Prozessen ausgeht, fallen alle wichtigen Entscheidungen unter den Tisch, die schon bei der Formalisierung getroffen werden. (Unsere neue Theorie der Programmierung sieht ja nicht einmal die Implementation als völlig formalen Prozess an, schon gar nicht die Spezifikation!)

2.2 Grenzeigenschaften: Korrektheit und Effizienz

Wir haben gesehen, dass sich die äußeren QM auf die Spezifikation beziehen und werden gleich sehen, wie sich die inneren QM auf den Programmcode beziehen. Bindeglied zwischen Spezifikation und Code ist die Eigenschaft *Korrektheit* einer Software. Diese ist kein Qualitätsmerkmal im eigentlichen Sinne, weil sie vom Anwender nicht festgestellt werden kann — dieser sieht nur Erwartungskonformität, Konsistenz (z.B. mit der Dokumentation), usw. Im Allgemeinen ist die Korrektheit von Software nur ein Spezialfall der Konsistenz zweier Entwicklungsdokumente.

Wenn man sich aber mit Software-Konstruktion beschäftigt, dann ist die Korrektheit die Leiteigenschaft, die alle äußeren QM subsumiert. Wenn man bei der Konstruktion auf ein Problem stößt, bei dem man sich auf ein äußeres QM bezieht (z.B. Robustheit oder Kompatibilität), so ist man eigentlich in die Analysephase zurück gerutscht: dort sollte man das Problem klären und zurückkommen mit einer genauen (und hoffentlich schriftlich niedergelegten) Vorstellung davon, was man als korrekt ansehen möchte. (Solche Rücksprünge sind — auch wenn man sie mit strukturiertem Vorgehen minimieren sollte — ganz normal und man sollte darauf vorbereitet sein.)

Die Effizienz als äußeres QM spielt eine Sonderrolle, erstens weil sie nicht auf funktionale Eigenschaften reduziert werden kann (wie das mit anderen äußeren QM geschieht), zweitens weil die Zeit-Anforderungen oft gar nicht spezifiziert werden. In anderer Hinsicht ist sie anderen Merkmalen aber ähnlich: sie hängt wesentlich von den Komponenten (Datenbank, XML-Parser, ...) aus dritter Hand ab (eine Frage, auf die ich hier gar nicht eingehe), und sie wird oft als Argument in falschen Argumentationen eingesetzt. Mehr über die widersprüchliche Rolle der Effizienz in Abschnitt 5.3.1.

2.3 Innere Qualität

Bei den inneren Qualitätsmerkmalen kann man sehr gut zwischen Zielen und *Kriterien* unterscheiden. Die Merkmale, die ich am Kapitel-Anfang nannte, sind Ziele; sie beziehen sich auf den Zweck guter Programmierung, auf das, was wir mit den Programmen mal machen wollen: sie weiterentwickeln, anpassen, verbessern, Fehler beseitigen, und sie überhaupt erstmal entwickeln — denn ein guter Stil zahlt sich auch sofort aus, nicht erst später. Die Kriterien sind Eigenschaften der Software, die diese Ziele befördern. Es ist zu bemerken, dass die *Lesbarkeit* oder *Verständlichkeit* von Softwaredokumenten (inklusive Programmcode) ein Nebenziel ist, das alle anderen Ziele zu subsumieren scheint. Donald Knuth schreibt (ich nehme an in [27]):

“ The computer programs that are truly beautiful, useful, and profitable must be readable by people. So we ought to address them to people, not to machines. All of the major problems associated with computer programming — issues of reliability, portability, learnability, maintainability, and efficiency — are ameliorated when programs and their dialogs with users become more literate. ”

Bei der Begründung dieser Aussage kommt man schon auf die beiden wichtigsten Kriterien innerer Qualität: Es ist klar, wenn ich ein Programm ändern will (zu welchem Zweck auch immer), dann muss ich es zunächst einmal verstehen, es sollte also möglichst *einfach* sein. Und da ich meist nur an einer Stelle ändern möchte, will ich sicher sein, dass ich nicht auch noch alles andere verstehen muss: die Software sollte also möglichst *teilbar* oder *modular* aufgebaut sein. Interessanterweise schlägt die Modularität sofort wieder auf die Einfachheit zurück: wenn es reicht die Teile einzeln zu verstehen, dann kann man auch das Ganze leichter verstehen. Die Modularität lässt sich in Unterkriterien einteilen. Bertrand Meyer behandelt in [29] nur diese Modularitätskriterien und nicht alle inneren Kriterien. Ich diskutiere Modularität in diesem Abschnitt nach den anderen Kriterien.

Kriterien unterscheiden sich von Zielen dadurch, dass man ihr Zutreffen leichter überprüfen kann als bei den Zielen: Die Softwarequalitätssicherung misst zum Beispiel die Verständlichkeit eines Softwareelements an der durchschnittlichen Einarbeitungszeit eines neuen Mitarbeiters (eine Messung, die man nur selten durchführen kann, und schon gar nicht oft genug, um einen sinnvollen Durchschnittswert zu erhalten). Oder sie misst die Korrigierbarkeit am durchschnittlichen Aufwand für eine Fehlerbeseitigung (das ist schon etwas realistischer, obwohl man natürlich passende Vergleichsdaten braucht, um diese Werte zu interpretieren). Alle diese Messungen können nur nach Fertigstellung des zu bewertenden Programmteiles geschehen, deswegen sind Messungen auch analytische Maßnahmen. Im Gegensatz zu diesen aufwändigen Prüfungen

der Ziele, kann man Kriterien schon während der Arbeit selbst prüfen und konstruktiv beim Abwägen von Alternativen einsetzen.

Neben Einfachheit und Teilbarkeit sind *Redundanzfreiheit* und *informale Konsistenz* die beiden anderen wichtigen Kriterien. Redundanzfreiheit bezieht sich natürlich auf die böse Redundanz: keine Information soll unnötig vervielfältigt werden, wir haben das ja in der Einleitung schon gesehen. Die informale Konsistenz verlangt, dass ähnliche Fragen auch immer ähnlich beantwortet werden. Die Konsistenz lässt sich auf vielen Ebenen der Software anwenden: beginnend bei einer einheitlichen Einrückung und Quellcodeformatierung, über ein einheitliches Namensformat für alle Arten von Bezeichnern, weiter über einheitliche Parameterübergabekonventionen und Seiteneffekte, bis hin zu informaleren Aspekten wie der Behandlung bestimmter Angelegenheiten wie Speicherverwaltung oder Identität. Wichtig bei der Konsistenz ist das Wort *ähnlich*, welches ganz gezielt informal gewählt wurde. Hätten wir *gleich* statt *ähnlich*, wäre das Redundanz, die zu beseitigen ist. Könnten wir die Relation *ähnlich* genauer spezifizieren, so würden wir einen Aspekt, der gleich ist, und einen Aspekt, der unterschiedlich ist, herausfinden — aber genau das ist die Voraussetzung der Anwendung von Abstraktion! Konsistenz ist also immer etwas Geschmacksache und arbeitet mit Analogien, die wir nicht mit einer formalen Methode erfassen können.

2.4 Grenzen der Unabhängigkeit

Die oben erwähnte Modularität kann man aus zwei Perspektiven betrachten: einmal wollen wir die Gesamtfunktionalität unserer Software in Angelegenheiten aufteilen, die das Gesamte vollständig abdecken; zum anderen wollen wir die einzelnen Module möglichst unabhängig voneinander verstehen. Hinter dieser *Unabhängigkeit* der Module steckt eigentlich das ganze Geheimnis eines guten Entwurfs. Wenn man sich die Theorie abstrakter Datentypen und abstrakter Maschinen anschaut, so werden dort Module spezifiziert, die in der Tat völlig unabhängig von den anderen Modulen sind, mit denen sie später mal eingesetzt werden. Die Spezifikation eines Stacks ist völlig eigenständig, und wenn man ihn mit Zeigern oder einem Array implementiert, so kann man diese Implementierungen auch völlig unabhängig voneinander verstehen. Abstrakte Datenstrukturen wie Mengen, Abbildungen und Prioritätswarteschlangen zeigen, dass man in der Tat eine gehörige Menge nicht-trivialen Codes hinter einer einfachen Spezifikation verstecken kann. Auch die strukturierten und funktionalen Programmiersprachen kann man mit ein paar einfachen Axiomen verstehen, ohne etwas über die zu Grunde liegenden Stacks, Heaps, Register und Maschinenbefehle zu wissen. (Siehe “axiomatische Spezifikationen” im Kapitel über Formale Theorien.)

Leider skaliert diese strenge Art von Unabhängigkeit nicht auf große Module und Komponenten einer Software. Man muss sich nur mal die Software eines Online-Buchladens anschauen: da gibt es Module für die Darstellung des Katalogs, dann die Darstellung des Einkaufswagens, persönliche Seiten für jeden Kunden, dann das “Backend”: ein Modul für die Lagerschnittstelle und den Versand, ein Modul für Kreditkartenzahlungen und Lastschriften, usw. usf. Zunächst könnte man meinen, das lässt sich alles schön hierarchisch gliedern: der Katalog als Spitze, wenn man auf “kaufen” klickt, kommt man zum Warenkorb; der Katalog kann auch persönliche Informationsschnipsel vom entsprechenden Modul abrufen und in seine Seiten einbinden... alles fein. Aber man stelle sich mal vor, was zum Beispiel alles passieren muss, wenn ein Kaufvorgang abgeschlossen wird: das Buch soll im Briefkasten des Kunden landen, gleichzeitig eine Bestätigungsnachricht in seiner elektronischen Mailbox; dann ändert sich natürlich sofort der Katalog und zeigt noch mehr Bücher desselben Autors und Themas an, außerdem wird irgendwo Geld abgebucht, und woanders werden womöglich Treuepunkte zugebucht.

Dieses kleine Beispiel zeigt schon, dass man Komponenten nicht vollständig unabhängig voneinander spezifizieren kann. Die Programmspezifikation, bzw. die geforderte Funktionalität der Software geben eine minimale Abhängigkeit vor, die zwischen den Komponenten bestehen *muss*. Es ist Aufgabe der Softwareentwickler, diese minimale Abhängigkeit nicht noch weiter zu vergrößern.

Das Beispiel zeigt auch schon einen ersten Weg, wie man die Teile einer Software bestimmen kann: man nimmt die von außen sichtbaren Komponenten, die hoffentlich schon in der Anforderungsanalyse als solche bestimmt wurden. Eine weitere Möglichkeit neue Einzelteile zu bekommen entspringt aus der Beseitigung von Redundanz: man schaut sich an, was die redundanten Stellen für eine Gemeinsamkeit haben und wo die Unterschiede sind. Dann versucht man durch *Abstraktion* die Gemeinsamkeiten zu isolieren. Ich erwähne das, weil diese Abstraktion einen Konflikt zwischen den Kriterien Einfachheit und Redundanzfreiheit hervorrufen kann. Solange man Redundanz noch mit einfachen Routinen beseitigen kann ist das natürlich noch keine Gefahr für die Einfachheit des Programms — im Gegenteil. Aber man kann schon mit Prozeduren mit vielen Parametern oder mit Prozedurparametern oder mit größeren Seiteneffekten oder wechselseitiger Rekursion eine große Komplexität einbauen — und mit Objekten, geschweige denn Vererbung eine noch viel größere. Um zu testen, ob einige so entstandene Programmteile zu abstrakt sind, kann man versuchen, deren Funktion wirklich komplett eigenständig zu beschreiben.

Die Einfachheit ist letztlich das wichtigste Kriterium. Man sollte sich bewusst machen, dass Einfachheit wirklich eine Operationalisierung des Zieles Lesbarkeit ist, und nicht nur ein anderer Name dafür. Einfachheit lässt sich oft an der bloßen Größe und an der Struktur eines Softwareelementes (lies: Codeabschnitt) erkennen. Einfach heißt: möglichst kurz, möglichst wenig abstrakt, möglichst wenig verschachtelt, möglichst keine Rekursion, Schleifen usw. Natürlich stehen diese Eigenschaften miteinander im Widerspruch (z.B. braucht man Abstraktion, um Kürze zu erreichen) und im Widerspruch mit den funktionalen Anforderungen (am einfachsten ist das Programm, das nichts tut). Deswegen sollte man auch seine Lösung nur so einfach wie möglich wählen — und (nach Einstein) nicht einfacher.

Die inneren Qualitätsmerkmale, angeführt von der Verständlichkeit der Lösung, lassen sich auf folgende vier Kriterien zurückführen: Einfachheit, Teilbarkeit, Konsistenz und Redundanzfreiheit. Genauso wie uns die nicht-funktionalen äußeren Qualitätsziele helfen, aus funktionalen Zielen und Anforderungen eine Spezifikation zu erstellen, helfen uns diese Kriterien die Spezifikation zu einer Software zu formalisieren.

Hier ist zu bemerken, dass diese Kriterien auch auf die Spezifikation selbst angewendet werden können, allgemeiner sogar auf jedes software-technische Dokument. Im Folgenden werde ich die Kriterien aber auf die Implementierung spezialisieren. Weiterhin kann man als Unterkriterien noch Angemessenheit der Lösung, Explizitheit oder Direktheit der Lösung und Ausgewogenheit der Teile notieren. Angemessenheit ist eigentlich nur ein Synonym für Einfachheit, das die Relativität der Kriterien zu einer gegebenen Aufgabenstellung explizit macht. Die Direktheit hilft dem Leser leicht den Bezug zwischen Zweck und Lösung festzustellen. Wenn man etwas indirekt macht, muss man in jedem Fall den Zweck explizit in der Dokumentation erwähnen — dann ist die Lösung aber immer noch komplizierter als die direkte Variante. Abstraktion ist Feind der Einfachheit, weil sie fast immer Indirektion einführt. (Das kann man umgekehrt zu einer Regel machen: falls Du eine Abstraktion ohne den Preis von Indirektheit findest, greif zu!) Die Ausgewogenheit macht sich oft bemerkbar, wenn sie fehlt: wenn ein Teil der Lösung wesentlich größer und komplexer ist, als ein anderer, sollte man doch mal eine andere Aufteilung probieren.

Die Analogie trägt aber nicht was die Verfeinerung von inneren Qualitätsmerkmalen angeht. Wenn man hier konkrete Ziele angibt (z.B. für die Portabilität: es sollen verschiedene Datenbanksysteme unterstützt werden), dann spiegeln sich diese Ziele bereits in der Spezifikation wieder — sie sind also plötzlich Bestandteil der äußeren Qualität.

2.5 Grenzen der Wiederverwendbarkeit

Bevor ich nun konkret zeige, wie man *lesbare Software* erstellt, möchte ich kurz begründen, warum ich nicht mehr verspreche, als dass die Software besonders lesbar wird: Ich glaube mehr ist einfach nicht möglich.

Man kann Software nicht im ganz allgemeinen Sinn *erweiterbar* oder *wiederverwendbar* implementieren. Entweder ist eine spezielle Verwendung oder Erweiterung von Anfang an vorgesehen — dann steht das in der Spezifikation und man macht dafür einen angemessenen Entwurf — oder sie ist es eben nicht. In keinem Fall kann man mehr tun, als die Software so gut verständlich wie nur möglich zu machen: also einfach, modular, konsistent und redundanzfrei.

Diese Regel ist eine Warnung vor den Träumereien, die manche neue Techniken oder so genannte Methoden versprechen: zum Beispiel findet man oft die Überzeugung, dass Vererbung hauptsächlich dem Erweitern existierender Software dient ([8, 40], auch Meyer ist davon nicht frei), aber das ist einfach nur unrealistisch. Eine Software kann man nur für einen bestimmten *Zweck* schreiben und nur diesen Zweck kann sie erfüllen. Der Zweck kann sehr abstrakt sein, wie z.B. die Verwaltung relationaler Daten; dann ist die Software natürlich auch leicht wiederverwendbar. Aber allein schon daran, dass trotz der Verwendung fertiger relationaler Datenbank-Systeme Softwareprojekte noch groß und schwierig sind, sieht man, dass Zwecke eben *konkret* sind und nicht allgemein abzudecken.

Wenn man eine Entwurfsentscheidung trifft, dann geht es nun mal qualitativ in die eine oder andere Richtung, je nach der *konkret verlangten Funktionalität*. Wenn die Entscheidung unabhängig von dieser verlangten Funktionalität wäre, könnte man die Lösung ja in der Methode schon festschreiben. Und da die Entscheidung nun also von der Funktionalität abhängt, folgt, dass eine durch Erweiterung oder anderes Einsatzgebiet geänderte Funktionalität auch die Entscheidung in Frage stellen kann — und damit die Grundstruktur der gesamten Software. Wenn man sich damit abfindet, kann man ganz in Ruhe seine Implementierung auf die genannten Kriterien hin optimieren anstatt auf diffuse Erweiterbarkeit oder Wiederverwendbarkeit. Ich werde an entsprechender Stelle auch an Beispielen zeigen, wie man alle Entscheidungen anhand der Kriterien trifft und begründet.

Eine letzte allgemeine Regel sollte der Software-Entwickler immer im Kopf haben:

Man trägt zur Qualität und Produktivität bei, indem man die Software so änderbar macht wie möglich (gute Strukturierung), und indem man dafür sorgt, dass man sie möglichst selten ändern muss (gute Anforderungsanalyse, bewusstes Treffen von Entscheidungen).

Das ist ein Kernprinzip der Softwaretechnik und es ist scheußlich schade, dass dies nicht als Basis des Wissensstands benutzt wird. Man kann daran übrigens auch schon die Schwäche der so genannten “leichtgewichtigen Prozesse” festmachen: diese konzentrieren sich zur sehr auf die Änderbarkeit. Der Änderbarkeit aber sind natürliche Grenzen gesetzt, genauso wie man nicht alle Entscheidungen beim ersten Mal richtig treffen kann. Man muss sich einfach auf beides konzentrieren und das versuche ich in dieser Arbeit.

3 Formale Beschreibungen

Formale Beschreibungsmethoden werden damit beworben, dass sie eine mathematisch exakte und eindeutige Bedeutung haben — beweisbar jegliches Missverständnis ausgeschlossen. In Wirklichkeit kann man mit formalen Beweisen aber immer nur formale Eigenschaften von Beschreibungen erfassen: man kann zeigen, dass eine Beschreibung eine andere logisch impliziert oder dass zwei Beschreibungen äquivalent sind (sich gegenseitig implizieren), aber man kann nichts über die wirkliche Bedeutung der Beschreibungen sagen. Formalisten machen sich auch über die wirkliche Bedeutung keine Sorgen. Sie gehen davon aus, dass ihre Axiome schon irgendwie in der realen Welt wichtig sind, und dann zeigen sie, was alles aus diesen Axiomen folgt (oder auch nicht). Oft lassen sich solcherart erzielte Ergebnisse auch intuitiv auf die Realität zurück übertragen, aber in letzter Zeit sind Fälle aufgetaucht, wo Programme formal aus Spezifikationen hergeleitet wurden, mit Beweis natürlich, jedoch war man sich am Ende über die Bedeutung des Programms nicht im Klaren, weil man nicht einmal genau sagen konnte, was die Spezifikation eigentlich bedeutet!

Natürlich behaupten die Autoren von Spezifikationen und Spezifikationssprachen immer, dass ihre Produkte eindeutig sind, aber sobald noch jemand anderer Meinung ist, ist das ja bereits eine Zweideutigkeit.

Wenn man eine ganz einfache Sicht auf die Programmierung einnimmt, dann braucht man sich um die Bedeutung (Semantik) formaler Dokumente überhaupt nicht zu kümmern. Man kann sagen “Ein Programm ist, was es tut. Ob es das Richtige tut, kann man nur feststellen indem man beobachtet, was es tut,” und die Bedeutung des Programmtextes ist von keiner Bedeutung! Aber selbst in der Extremen Programmierung sagt man ja “The code is the design”, also muss dieser Text doch eine bestimmte Rolle spielen. Natürlich kann man sich bei Programmtexten die Bedeutung ganz einfach über die Ausführung definieren — dann muss man zwar manchmal etwas umständlich argumentieren, aber es geht.

In dieser Arbeit gehen wir aber davon aus, dass die moderne Programmierung auch Beschreibungen braucht, die nicht ausführbar sind, allen voran natürlich die Spezifikationen. Auch hier kann man sich wieder herausreden und die Semantik einer Spezifikation einfach über die Programme definieren, die diese Spezifikation erfüllen. Das erscheint praktisch, weil man zwischen Programmen und Spezifikationen ja sowieso eine formale Relation braucht, um erstere mit formaler Unterstützung zu entwickeln. In der Praxis ist eine solche umgekehrte Definition aber natürlich völlig kontraproduktiv: man möchte ja gerade über die Angemessenheit einer Spezifikation nachdenken können, ohne sich dabei in Implementierungsdetails zu verlieren.

Was wir in diesem Kapitel lernen, ist also wichtig, um allgemeine formale Beschreibungen (nicht nur Programme) zu verstehen. Außerdem wird es im nächsten Kapitel gebraucht, wenn wir formale Theorien der Programmierung betrachten. Das Material dieses Kapitels ist hauptsächlich den Arbeiten von Micheal Jackson (nicht dem Sänger) und Pamela Zave entnommen [24, 54, 22]. Diese Arbeiten sind selbst unter Wissenschaftlern noch nicht weit verbreitet und ihr genauer Bezug zur Praxis ist auch nicht recht klar. Bei meinem Versuch, sie zu erklären, habe ich einige neue Sichtweisen hervorgebracht. Einige davon, finden sich bereits in meinem Praktikumsbericht [45]. Es scheint auch noch keine deutsche Literatur dazu zu geben, abgesehen von Heisels Vorlesungsskripten [17], die aber nur eine schlichte Übersetzung von [54] enthalten.

3.1 Qualität von Beschreibungen, insbesondere Objektivität

Auf Beschreibungen ganz allgemein (ob nun formal oder informal, ausführbar oder nicht) treffen die gleichen inneren Qualitätsmerkmale zu wie auf Programme. Während aber Programme

durch ihre Ausführbarkeit immer eine exakte und eindeutige Bedeutung haben, trifft das auf andere Beschreibungen nicht zu. Programmierer müssen nur sicherstellen, dass die bereits eindeutige Bedeutung ihrer Programme für den Leser auch leicht erkennbar ist. Im allgemeinen Fall der (nicht unbedingt ausführbaren) Beschreibungen kommen zu den Kriterien der *Einfachheit*, *Modularität*, *Redundanzfreiheit* und *Konsistenz* noch hinzu: die *Eindeutigkeit* und *Objektivität*. Oft wird auch noch *Vollständigkeit* aufgezählt (klar: man soll nichts vergessen), aber manche Beschreibungen sollen ja ganz absichtlich unvollständig sein, sei es weil nichts Genaueres bekannt ist oder nicht gebraucht wird (man denke an die Angemessenheit) oder weil man dem Programmierer noch gewisse Freiheiten lassen will. (Letzteres ist das Argument der Formalisten für den Indeterminismus.)

Die Verfechter formaler Methoden behaupten, dass formale Beschreibungen immer objektiv und eindeutig sind, und dass informale Beschreibungen mit derselben Objektivität viel länger und schwieriger zu lesen sind. Bei einem mathematischen Anwendungsbereich trifft das sicher zu. Zum Beispiel ist $\text{verkaufspreis} = \text{nettopreis} * 1,15$ schon recht kurz. (Obwohl das halbformale $\text{verkaufspreis} = \text{nettopreis} + 15\%$ für die meisten Menschen verständlicher und ebenso eindeutig ist.) Dass aber formale Beschreibungen im Allgemeinen nicht unbedingt kürzer und schon gar nicht leichter lesbar sind, kann man sich leicht veranschaulichen indem man mal versucht eine beliebige technische Gebrauchsanweisung oder ein Küchenrezept zu formalisieren!

Es ist schwer zu sagen, bei welchen Beschreibungen genau sich Formalisierung lohnt. Ein gutes Zeichen ist es sicher, wenn in der Beschreibung mathematische Konzepte wichtig sind: Zahlen, Mengen, Funktionen, logische Operatoren... Auch ist es von Nutzen, wenn die zu formalisierende Beschreibung eine gewisse Redundanz aufweist, zum Beispiel in Form von häufig vorkommenden Begriffen, die am besten noch ein paar einfachen Gesetzen gehorchen. (So ist es zum Beispiel bei den Finanzgeschäften in [35].) In der Programmierung sind uns formale Ansätze natürlich nützlich, weil Programme selbst schon formal sind. Genaueres dazu im Kapitel "Allgemeine Theorie".

Leider sind mathematische Beschreibungen aber nicht immer automatisch so objektiv und eindeutig, wie es uns manch einer glauben machen will. Wir müssen uns nur einmal das Beispiel $\text{verkaufspreis} = \text{nettopreis} * 1,15$ ansehen und fragen: "Welches ist der Preis, den ich eigentlich bezahlen muss?" Und: "Welches ist der auf dem Produkt ausgezeichnete Preis?" Im Beispiel scheint der verkaufspreis relativ sicher der Preis zu sein, den ich bezahlen muss, aber ich könnte ja auch Großkunde oder sonstiger Rabattempfänger sein. Was die zweite Frage angeht, so könnte man ja die Beschreibung ändern zu $\text{verkaufspreis} = \text{ausgezeichneter_preis} * 1,15$, aber dann wäre immer noch nicht klar, ob das nun zum Beispiel der am Produkt ausgezeichnete Preis ist oder der am Regal oder der im Prospekt, oder der im Internet! (Selbst Aldi und Lidl haben ja Prospekte und Internetseiten mit ihren Gelegenheitsangeboten...)

Wir brauchen also zum Verständnis jeder formalen Beschreibung einer realen Sache auch noch informale Beschreibungen zur Begleitung. In den nächsten Abschnitten sehen wir welche.

Hieran kann man schon sehen, warum sich manche Sachverhalte so schlecht formalisieren lassen: man braucht einfach viel zu viele Begriffsbestimmungen pro formalem Textanteil. Viel zu viel Allgemeinwissen nötig.

3.2 Elementare Beobachtungen und Begriffsbestimmungen

Im allgemeinen Fall sind alle formalen Beschreibungen Bool'sche Ausdrücke; das ist zumindest die einfachste Sicht [13]. Graphischen Beschreibungen sieht man das nicht an, aber deren Semantik muss auch Bool'scher Natur sein. Eine Beschreibung muss immer *wahr* sein, für die Dinge die sie beschreibt und *falsch* für alle anderen. In diesen Ausdrücken kommen verschiedene Arten von Bezeichnern vor: einige stammen aus mathematisch-informatischer Tradition, so wie + oder \wedge , einige benutzen wir nur für die aktuelle Anwendung. Eine ganz wichtige Unterscheidung ist die folgende: wenn ich eine Beschreibung auswerte, also für ein bestimmtes Ding prüfe, ob es

von der Beschreibung beschrieben wird, so wende ich manche der Begriffe in der Beschreibung auf das Ding an, und andere nicht.

Wenn zum Beispiel die Gleichung $sportpuls = 150 - lebensalter$ einen Sportler in gutem Trainingszustand beschreibt, so kann ich sie auf mich anwenden, indem ich meinen Puls beim Sport und mein Alter einsetze, aber für die Symbole “-”, “=” und “150” setze ich nichts ein, sie haben ihre übliche mathematische Bedeutung. In diesem Beispiel sieht das noch sehr einfach aus, aber im nächsten kommt es darauf an. Nehmen wir obige Formel als Definition für den $sportpuls$ und die Beschreibung $puls > sportpuls + 10$. Hier muss ich meine Werte plötzlich für die Begriffe $puls$ und $lebensalter$ einsetzen, um die Beschreibung auf mich anzuwenden.

Die Frage ist ja: “Ist mein Puls größer als mein Sportpuls plus 10?”, wobei “Sportpuls” für “150 - Lebensalter” steht. Also ist die Frage: “Ist mein Puls größer als 160 minus mein Lebensalter?”.

In diesen Beispielen wurde stets aus dem Zusammenhang heraus klar, welche Begriffe auf die Realität bezogen werden sollen, und welche nicht. Wenn man aber große Beschreibungen hat, kann man sich nicht mehr darauf verlassen. Außerdem haben wir ja an dem Preisbrecher-Beispiel schon gesehen, dass auch nicht immer klar ist, auf welche Elemente der Realität sich ein Begriff denn nun genau bezieht. Was wir also brauchen ist für jede Beschreibung eine Liste ihrer *externen Begriffe* zusammen mit einer *Begriffsbestimmung* für jeden dieser Begriffe. Eine Begriffsbestimmung würde man umgangssprachlich als Definition bezeichnen, aber wir wollen als “Definition” nur formale Aussagen bezeichnen, deswegen brauchen wir den speziellen Begriff “Begriffsbestimmung”. Der Unterschied zwischen beiden ist sehr wichtig und wird gleich noch genauer erklärt.

Alle anderen Begriffe sind *intern*, sie sind entweder durch den gewählten Formalismus bestimmt (in den bisherigen Beispielen die Arithmetik) oder sie wurden explizit formal *definiert* (so wie der $sportpuls$ eben). Bisherige Arbeiten von Jackson stellen auch den Unterschied zwischen definierten (*defined*) und bestimmten (*designated*) Begriffen in den Vordergrund. Diese Unterscheidung ist aber besonders schwierig, wenn man mit informalen oder halbformalen Beschreibungen arbeitet: Definitionen sind dann nicht unbedingt in formaler Sprache und unterscheiden sich also darin nicht mehr von den Begriffsbestimmungen. Die Unterscheidung aber, für welche Begriffe einer Beschreibung wir Werte aus der Realität einsetzen sollen und für welche nicht, ist immer von Bedeutung. In einer Arbeit [24] sprechen Jackson und Zave auch von einem *identification set* einer Beschreibung als der Menge von externen Begriffen (mit Bestimmungen).

Es gibt noch ein anderes Kriterium mit dem man Begriffsbestimmung von Definitionen unterscheiden kann, wenn letztere nicht in formaler Sprache geschrieben sind. Begriffsbestimmungen schreibt man so, dass sie für Personen mit Sachkenntnis im Anwendungsbereich der Spezifikation eindeutig sind. Wie genau man dabei sein muss, hängt also von der Anwendung ab. Definitionen allerdings sind Aussagen, die nur bereits bestimmte oder zu definierende Begriffe enthalten. Eine Definition soll zu ihrem eindeutigen Verständnis aber kein Allgemeinwissen voraussetzen (sondern nur Kenntnis der Sprache). Während man also Begriffsbestimmungen naturgemäß nicht formalisieren kann, sollten Definitionen prinzipiell immer formalisierbar sein.

Wenn man eine Menge von externen Begriffen einer Beschreibung festgelegt hat, ergibt sich automatisch, welche Aussagen eigentlich Definitionen sind und welche *falsifizierbare Behauptungen*. Letztere enthalten nur Begriffe, die man direkt an der Realität überprüfen kann, also externe, begriffsbestimmte Begriffe. Dies ist eine strenge Folgerung: jede Aussage, die einen unbestimmten Begriff enthält, wird automatisch zum Teil seiner Definition, ganz einfach weil man die Aussage als Behauptung gesehen ja niemals überprüfen könnte. Dieser Definitionsautomatismus ist übrigens sehr praktisch, weil viele formale Sprachen (zum Beispiel Zustandsautomaten) es nur eingeschränkt oder überhaupt nicht erlauben neue Begriffe zu definieren.

Um wirklich objektiv überprüfbare Behauptungen aufzustellen, müssen die externen Begriffe so einfach wie möglich sein. Deswegen spricht man auch von *elementaren Beobachtungen*. Wir haben das schon am Beispiel des Preises von Waren gesehen — woran erkennt man genau, welches der Preis ist? Ein anderes Beispiel (aus [24]) ist der Begriff “Kunde”. Ist das jemand,

der Prospekte erhält, oder jemand, der Dinge bestellt, oder jemand, von dem Geld eingeht? Am besten ist es, solche elementaren Beobachtungen zu bestimmen und dann den Begriff “Kunde” mit Hilfe dieser Beobachtungen zu definieren. Mit der Definition hat man dann gleich eine erste Aussage, die man formal verwenden kann. Dieses Vorgehen entspricht einer Domänenanalyse, die etwas tiefer geht, als einfach den “Kunden” als Basisbegriff zu benutzen. Man erhält mehr Material.

Rein mathematisch gesehen sind Begriffsbestimmungen wertlos — man kann daraus keine Folgerungen ziehen! Anders bei Definitionen: wenn ich sage “sei $t = 6$ ” dann kann ich auch sagen “also ist $t + 1 = 7$ ”. Wenn ich aber einen Begriff bestimme — “Sei t die Zeit in Sekunden, die ich zum Aufspülen brauche,” — folgt daraus noch nicht einmal $t \in R$ oder $t \geq 0$; beides muss ich separat als Aussage angeben. Aus formaler Sicht bilden all die Aussagen, die ich über einen informal bestimmten Begriff mache, eine axiomatische Definition dieses Begriffs. So werden formale Systeme fast immer aus ein paar Primitiven aufgebaut, die axiomatisch definiert sind, und dann kann man andere Begriffe durch direkte Definitionen hinzufügen. Begriffsbestimmungen helfen dabei einerseits die indirekt definierten Begriffe besser zu verstehen (eine in sich geschlossene Kurzbeschreibung statt mehrerer unabhängiger Axiome), andererseits sind sie von entscheidender Bedeutung bei der Verbindung von formaler Welt und Realität. (Logiker finden Definitionen auch gut, weil man damit im Gegensatz zu Axiomen keine logischen Widersprüche einführen kann.)

Zusammenfassung für die Praxis: Man mache sich bewusst, welche Begriffe elementare Beobachtungen bezeichnen und welche Definitionen. Dies kann man in einer einfachen Liste dokumentieren. Sollte es bei der Erstellung oder Benutzung der Beschreibung dann Verständnisschwierigkeiten geben, kann man die Begriffe bewusst hinterfragen und gegebenenfalls die Liste an neue Erkenntnisse anpassen. Ohne explizite Angabe externer Begriffe fehlt einer (formalen oder informalen) Beschreibung nicht nur eine wichtige Information, sondern man kehrt auch die Probleme unter den Tisch, die bei der Interpretation der Beschreibung auftreten können.

3.3 Eine kleine Phänomenologie

Michael Jackson, der Philosophie studierte bevor er Informatiker wurde, bezeichnet die elementaren Beobachtungen als “Phänomene” und spricht wie in der Logik üblich von “Prädikaten” an Stelle von “Bool’schen Ausdrücken”. In der Tat sind beide Sichtweisen nützlich: formal gesehen brauchen wir nicht zwischen Bool’schen und anderen Ausdrücken zu unterscheiden, die Regeln zur Manipulation sind die gleichen. Die philosophische Sicht ist aber nützlich, wenn wir formale Beschreibungen *interpretieren* wollen, beziehungsweise schlicht und einfach an der Realität *überprüfen*. Dann macht es durchaus Sinn, zwischen *Prädikaten* und *Termen* (d.h. nicht-Bool’schen Ausdrücken) zu unterscheiden.

Aus formaler Sicht sind alle Begriffe in einem Ausdruck gleichwertig, ihre Bedeutung ergibt sich nur aus ihrem Vorkommen in diesem und anderen Ausdrücken. In der Anwendung von formalen Beschreibungen lohnt es sich aber die Unterscheidung zu machen, weil je nach der “Art” eines Begriffs bei der Formalisierung andere Aspekte zu beachten sind. Im Folgenden gebe ich die Phänomenologie aus Michael Jacksons Buch “Problem Frames” wieder.

Begriffe bezeichnen Phänomene (engl. *phenomena*). Terme stehen dabei für Individuen (*individuals*) und Prädikate für Beziehungen zwischen den Individuen (*relations*). Individuen können sein: *Werte*, *Dinge* und *Ereignisse* (engl. *values*, *entities*, *events*). Für die Individuen ist wichtig, dass man sie immer genau unterscheiden kann. Dafür müssen die Begriffsbestimmungen sorgen. Werte sind unveränderliche, ungreifbare Einheiten mathematischer Natur; also genau das womit Computer rechnen. Die wichtigsten Arten sind Zahlen und Zeichenketten. Dinge hingegen sind greifbar und veränderlich, wie die Objekte der realen Welt. Wenn Computer Informationen über Dinge verarbeiten, so muss das immer mittelbar über Werte (Eigenschaften der Dinge) geschehen.

Ereignisse brauchen wir in dieser Arbeit nicht, deswegen gehe ich nicht weiter darauf ein.

Es sei aber bemerkt, dass gerade für Ereignisse viele Dinge zu beachten sind, wenn man die formalen Begriffe auf die Realität anwenden will.

Beziehungen können sein: *Zustände*, *Rollen* und *Wahrheiten* (engl. *states*, *roles* und *truths*). Zustände verbinden Dinge untereinander (so wie die Beziehungen in ER-Diagrammen) oder mit Werten (dann sind es *Eigenschaften*). Zustände können sich mit der Zeit verändern. Rollen bezeichnen die Teilnehmer von Ereignissen und wir ignorieren sie hier. Wahrheiten verbinden Werte untereinander und wie der Name schon sagt, sind sie immer wahr (so wie die Aussage $1 + 1 = 2$).

Die Unterscheidung der Phänomen-Arten hat sich aber leider auch noch nicht durchgesetzt und man macht es gerade falsch herum: viele Anwendungen von formalen Methoden gehen nicht auf die Unterschiedlichkeit von Begriffsarten ein; insbesondere Ereignisse leiden darunter, dass man sie wie einfache Werte interpretiert. Auf der anderen Seite unterscheiden noch viele formale Systeme zwischen Prädikaten und Termen, obwohl man erstere viel leichter als Bool'sche Ausdrücke behandeln könnte und damit genau wie die Terme auch.

3.4 Fallstudie Eins: Mitarbeiterverfügbarkeit, logisch

In diesem Abschnitt wollen wir uns mal anschauen, wie man einen einfachen Sachverhalt formalisieren kann und wie man verschiedene halb-formale Notationen auf eine gemeinsame formale Basis stellt.

Klaus kann nur mittwochs und donnerstags; Peter kann nur montags, mittwochs und freitags; Franz kann immer, wenn Klaus nicht kann (z.B. weil er ihn vertreten muss); Gabi kann nur an ungeraden Tagen; Claudia kann nur, wenn Gabi oder Franz auch kann (z.B. wenn sie eine Mitfahrgelegenheit braucht). An welchen Tagen sind mindestens drei Mitarbeiter verfügbar?

Wenn wir nur das Rätsel lösen sollen, reicht eigentlich jede beliebige Formalisierung aus, und man kann natürlich auch ganz informal argumentieren. Wenn wir aber ein Termin-Planungsprogramm schreiben wollen, dann brauchen wir bessere Grundlagen. Eine erste Idee mag sein, einfach eine Tabelle zu erstellen:

Mitarbeiter	Montag	Dienstag	Mittwoch	Donnerstag	Freitag	Samstag
Klaus			kann	kann		
Peter	kann		kann	kann	kann	
...						

Man kann auch leicht die restlichen Aussagen in diese Tabelle übertragen. Für manche Anwendungen mag das sogar schon genug sein! Wir wollen aber einmal annehmen, dass unser Programm auch Abhängigkeiten berücksichtigen soll, so dass eine Änderung von Klaus' Verfügbarkeit sich auch auf diejenige von Franz auswirkt. Hierzu könnten wir einfache Formeln einführen: $franz \equiv \neg klaus$, $claudia \equiv (gabi \vee franz)$. (Man kann daraus übrigens schon folgern: $claudia \equiv (klaus \Rightarrow gabi)$.) Dann ergeben sich aber zwei Probleme: erstens müssen wir den Zusammenhang von Tabelle und Formeln überlegen. Das tun wir am besten, indem wir elementare Begriffe bestimmen und beide Formen auf Aussagen über diese Begriffe übersetzen. (Übrigens benutzen Hoare und He [21] genau dieselbe Technik, um verschiedene Theorien der Programmierung in Einklang zu bringen. (Siehe Kapitel über "Formale Theorien".) Näher an unserem Beispiel sind Jackson und Zave [53, 52], die diese Technik ebenfalls benutzen, um verschiedene Spezifikationen gemeinsam zu verwenden.) Das zweite Problem ist, dass allgemeine Bool'sche Ausdrücke möglicherweise schon zu ausdrucksstark sind, um von Computern (anders als durch einfache Auswertung) bearbeitet zu werden. Man denke an das Erfüllbarkeitsproblem. Dieses Problem löst man durch geeignete Einschränkungen: was ist minimal für die Erfüllung der Anforderungen nötig?

Als semantische Basis führen wir also ein Prädikat $kann(ma, tag)$ ein, mit der Bedeutung “Der Mitarbeiter ma ist am Wochentag tag verfügbar.” Ein “kann” in Zeile ma und Spalte tag unserer Tabelle führt dann zur Aussage $kann(ma, tag)$. Die Bool’schen Formeln übersetzen wir, indem wir ein $\forall tag$ davor schreiben und jedes Vorkommen eines Mitarbeiter-Namens ma ersetzen durch $kann(ma, tag)$. (Hier sehen wir, dass es problemlos möglich ist, den Tag in der Formel zu erwähnen. “Franz kann montags und wenn Klaus nicht kann.” wird zu $\forall tag.kann(franz, tag) \equiv tag = Montag \vee \neg kann(klaus, tag)$. Nur mal, um zu zeigen, dass “und” informal nicht immer “und” formal bedeutet...)

Für die Beispiele oben reicht es auch aus, ganz einfache Einschränkungen zu machen: (1) alle Formeln haben die Form $ma \equiv \dots$ (oder allgemein $\forall tag.kann(ma, tag) \equiv \dots$), (2) Die Mitarbeiter, die in der Tabelle vorkommen und die, die auf der rechten Seite von einer Formel stehen, sind verschieden, und (3) man kann die Formeln so sortieren, dass jede Formel sich nur auf Mitarbeiter bezieht, die in der Tabelle oder schon darüber definiert wurden. (Übrigens, kleiner Vorgriff auf das nächste Kapitel: wenn wir die Bool’sche Negation verbieten, sind die rechten Seiten der Formeln monoton. Wir können dann Rekursion zulassen (also Einschränkung (3) weglassen) und mit kleinsten Fixpunkten arbeiten. Diese Eingabe-Sprache ist dann äquivalent zu PROLOG.)

Jetzt, da die Semantik unserer Beschreibungen festliegt, können wir uns auch noch andere Formen ausdenken. Zum Beispiel suggeriert die tabellarische Form, dass man einfach die Bool’schen Ausdrücke in die Zellen aufnimmt — so wie in einer Tabellenkalkulation. Die Quantifizierung über alle Wochentage erreicht man dann einfach, indem man Formeln für mehrere Zellen auf einmal einträgt. Die semantische Einfachheit dieses Modells wird also geschmälert durch die Redundanz, die von diesen Kopien erzeugt wird.

3.5 Einschränkungen und Kontrolle

Bisher haben wir immer nur einzelne, unabhängige Objekte beschrieben. Wenn man aber mit Systemen zu tun hat, die aus mehreren Komponenten bestehen, dann braucht man noch zwei weitere Meta-Informationen, um formale Beschreibungen interpretieren zu können. Im einfachsten Fall haben wir immer zwei Komponenten: die zu erstellende Software und ihren Anwendungsbereich (*application domain*). In jeder nicht-trivialen Anforderungsanalyse brauchen wir nicht nur Beschreibungen was der Computer tun soll, sondern auch Computer-unabhängige Beschreibungen des Anwendungsbereichs. Viele formale Methoden versagen hier bereits, weil sie beide Beschreibungen in einer einzigen “Spezifikation” mischen.

Im allgemeinen Fall brauchen wir folgende Meta-Informationen:

Einschränkung Welche *Komponente* eines Systems ist für die Einhaltung einer *Beschreibung* verantwortlich?

Kontrolle Welche *Komponente* kann eine *elementare Beobachtung* beeinflussen?

Sichtbarkeit Welche *elementaren Beobachtungen* werden von einer *Komponente* (direkt) wahrgenommen?

Im einfachen Fall eines Computers und seines Anwendungsbereichs als Komponenten haben wir die *gegebene* (indikative) Beschreibung des Anwendungsbereichs und die *geforderte* (imperative, optative) Beschreibung des Computerverhaltens. Der Witz dabei ist, dass wir zur Beschreibung des Computerverhaltens elementare Beobachtungen aus dem Anwendungsbereich verwenden, die nicht unbedingt vom Computer kontrolliert und nicht einmal wahrgenommen werden! Auf diese Weise spezifizieren wir das Computerverhalten *indirekt*, während die Anforderungen auf ganz natürliche Weise *direkt* wiedergegeben werden, unabhängig davon wie der Computer sich später einmal in den Anforderungsbereich einfügen wird.

Betrachten wir einmal die Steuerung eines Thermostats. Sei *gewuenscht* die vom Benutzer am Thermostat eingestellte Temperatur, und *temp* die aktuelle Raumtemperatur. Diese Begriffsbestimmungen sind hier eine ausreichende Annäherung, wir müssen nicht dazu sagen, wo genau die Temperatur gemessen werden soll. Die Anforderung an das Thermostat ist dann ganz grob gesagt $temp = gewuenscht$. Diese Beschreibung ist eine Einschränkung an die Software, obwohl diese nur ganz eingeschränkt Einfluss auf die elementaren Beobachtungen nehmen kann.

Man kann an diesem Beispiel übrigens auch schön sehen, wie man durch weitere Annäherungen aus der groben Beschreibung eine feinere gewinnt. Zunächst führen wir eine Konstante ϵ ein, die später festgelegt wird. Dann können wir beschreiben $gewuenscht - \epsilon \leq temp \leq gewuenscht + \epsilon$.

Auch diese Aussage ist noch keine Spezifikation, aber wir können uns weiter annähern, indem wir den Begriff *heizen* definieren mit

$(temp < gewuenscht - \epsilon) \Rightarrow heizen \Rightarrow (gewuenscht + \epsilon < temp)$.

Diese "Bool'sche Ungleichung" beschränkt *heizen*, falls die Temperatur außerhalb des gewünschten Intervalls liegt. Für nicht an formale Logik gewöhnte Menschen ist vielleicht die folgende äquivalente Aussage leichter zu verstehen:

$$temp < gewuenscht - \epsilon \Rightarrow heizen \\ \wedge gewuenscht + \epsilon \geq temp \Rightarrow \neg heizen$$

In diesem Beispiel haben wir noch keine explizite Beschreibung des Anwendungsbereichs gebraucht. Da wir sowieso nur mit Annäherungen arbeiteten, haben wir uns nur informal überlegt, dass jeder Schritt noch dem originalen Ziel zuträglich ist. Wenn wir das Beispiel weiter führen, stoßen wir auch auf Anwendungswissen (d.h. Beschreibungen des Anwendungsbereichs).

Sei *gas* die Beobachtung "Gas strömt in den Brenner der Heizung" und *feuer* "die Heizung brennt (natürlich nur innen)". Dann haben wir die Aussage $feuer \Rightarrow gas$. Bisher waren alle elementaren Beobachtungen für die Steuerungssoftware sichtbar, aber wurden nicht direkt von ihr kontrolliert. Jetzt fügen wir hinzu: *gas* wird von der Software kontrolliert, *feuer* ist nur für sie sichtbar. Damit hat die Software jetzt indirekt die Möglichkeit das Feuer auszuschalten.

Bei in der Praxis verwendeten Formalismen sind die Eigenschaften Sichtbarkeit/ Kontrolle/ Beschränkung oft implizit (durch die Methode) vorgegeben (klassisches Beispiel: Vor- und Nachbedingungen). Bei anderen Formalismen sind sie a priori überhaupt nicht festgelegt (Beispiel: CSP), was dazu führt, dass manche (oder viele?) Anwendungen dieser Formalismen praktisch völlig unfundiert sind.

Michael Jackson hat die einfache Notation von *Problemdiagrammen* vorgeschlagen, um die Meta-Informationen zu notieren. Komponenten sind darin Rechtecke, Beschreibungen sind Ovale. Sichtbare Phänomene werden an einfachen Linien zwischen den Komponenten untereinander und den Beschreibungen aufgezeichnet. Mit dem Ausrufezeichen "!" wird spezifiziert, welche Komponente ein Phänomen kontrolliert und schließlich zeichnen wir einen einfachen Pfeil von einer Beschreibung zur von ihr eingeschränkten Komponente. Für den einfachen Fall von Computer und Anwendungsbereich erhalten wir das Diagramm aus Abbildung 3.1 (zitiert aus [45]).

Jackson benutzt den Begriff "Domäne" für Komponenten außerhalb der Software. Dies können Menschen, Maschinen oder Daten sein. In seinem Buch "Problem Frames" [22] und anderen Publikationen [23] untersucht er die verschiedenen Angelegenheiten (*concerns*), die bei der Formalisierung zu beachten sind. Im Folgenden verwenden wir formale Beschreibungen aber nur innerhalb der Software, so dass wir meist von solchen Sorgen frei sind. Bei einem einfachen Programm ist klar, dass der Vorherzustand vom Programm beobachtet, aber nicht kontrolliert wird, und dass das Programm den Nachherzustand kontrolliert. "Logischerweise" schränkt dann die Vorbedingung immer den Aufrufer ein und die Nachbedingung das Programm selbst.

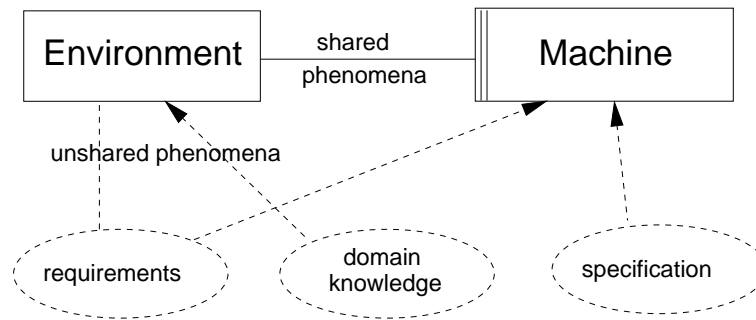


Abbildung 3.1: Domänen und ihre Beschreibungen

3.6 Fallstudie Zwei: Diplomacy

An diesem Beispiel sehen wir die große Bedeutung von Definitionen zur Strukturierung von Beschreibungen. Zum selben Ergebnis kommen auch Jackson und Zave in [52, 53].

Diplomacy ist ursprünglich ein Brettspiel, man spielte es auch sehr gern per Post, in der Neuzeit natürlich per elektronischer Post übers Internet. Das Besondere an Diplomacy im Gegensatz zu anderen Spielen ist, dass es keine Runden gibt (im Sinne von “ich bin dran, Du bist dran, ...”), sondern alle Spieler machen ihre Züge gleichzeitig und ein Spielmeister (oder Richter) wertet dann das Ergebnis dieser Züge aus. Dies ist notwendig, weil das Ergebnis eines Zuges von den Zügen der anderen Spieler abhängt, dadurch wird das Spiel auch besonders interessant. Aufgabe der Diplomacy-Regeln ist es grob gesagt zu gegebener Stellung und Spielzügen eindeutig eine neue Stellung zu bestimmen. Dazu müssen die Regeln so einfach beschrieben sein, dass jeder Spieler sich die Wirkung seiner Züge vorstellen kann. Außerdem verwendet man heutzutage Computer als Richter, und für diese müssen die Regeln dann natürlich auch so beschrieben sein, dass sie selbst in den extremsten Extremfällen ein eindeutiges Ergebnis liefern. In einem normalen Spiel kommen bei weitem nicht alle diese Fälle vor, so dass ein menschlicher Richter auch mit gesundem Halbwissen und ein bisschen Sinn für Gerechtigkeit gut arbeiten kann. Ein Computer kann das aber nicht.

Die offiziellen Regeln des (Brett-)Spiels waren schon immer formal unvollständig, deswegen gibt es verschiedene Anstrengungen, diese Regeln zu vervollständigen. Man kann alle Versuche der Regelbeschreibung in drei Gruppen einteilen:

Regel-basiert Die Regeln des Brettspiels sind als eine Reihe von Wenn-Dann-Aussagen formuliert, wobei man implizit annimmt, dass Regeln mit spezifischerem Wenn-Teil Vorrang vor allgemeineren Regeln haben. Wenn man den Vorrang aber streng interpretiert, sind die Dann-Teile zu schwach, um eine eindeutige Lösung zu bestimmen. Zum Beispiel gibt es die Regel Frontalschlacht: “Zwei Einheiten können nie ohne Geleitzug ihre Räume tauschen.” Diese Regel sagt nicht, was die Einheiten denn stattdessen machen sollen. Außerdem ist es bei dieser Art der Beschreibung schwer, eine Lösung zu finden, selbst wenn sie eindeutig bestimmt ist. Man muss eine Reihenfolge der Regelanwendung finden, und das ist oft nicht leicht.

Algorithmisch Diese Form löst beide genannten Probleme auf einmal: Die Regeln werden als Algorithmus spezifiziert, den man der Reihe nach abarbeiten kann. Dieser Algorithmus ist im günstigsten Fall wirklich immer ausführbar und damit eindeutig. Die bisher publizierten Algorithmen sind aber alle viel komplizierter als nötig. Außerdem ist eine algorithmische Spezifikation weniger flexibel: einerseits um verschiedene Varianten der Regeln zu diskutieren (und davon gibt es *sehr* viele), andererseits um verschiedene Implementierungen zu erlauben (zum Beispiel auch solche, die den Entscheidungsprozess graphisch animieren oder andere Zwischenergebnisse ausgeben).

als Beispiel-Sammlung Es gibt in der Tat eine große Beispiel-Sammlung für Stellung, Züge und zugehörige Lösungen im Internet, in deren Vorwort steht: “Die Regeln kann man sowieso nicht vollständig, eindeutig und gleichzeitig einfach angeben, also geben wir für strittige Situationen hier die Lösungen an.” Die Idee ist simpel: der menschliche Spieler kann sich aus Betrachtung der Beispiele ein mentales Modell der Regeln bilden, und sollten sich doch mal zweie uneinig sein, fügt man den strittigen Fall einfach als neues Beispiel mit ein. Außerdem dient die Beispiel-Sammlung auch dem automatischen Test von Computer-Richtern.

Übrigens sind auch die beiden ersten Regel-Formen, insbesondere die erste, auf Beispiele angewiesen, um sich verständlich zu machen! Manchen algorithmischen Beschreibungen muss man zugestehen, dass sie schon guten Gebrauch von Definitionen machen (wenn auch oft nur in Form von Prozeduren) und dadurch Struktur erhalten. Außerdem bestimmt der Algorithmus einige Variablen, die zusammen mit dem Ergebnis als dessen “Erklärung” ausgegeben werden können; auch das ist schon ein Fortschritt für die Transparenz der Regeln. Unsere neue Beschreibung liefert solche Erklärungsinformationen über die definierten Begriffe und macht damit den Entscheidungsprozess (bzw. dessen Ergebnis) noch transparenter. Ich habe diese Beschreibung übrigens gewonnen, während ich die Regeln in der mathematischen Spezifikationsprache B formalisierte. Diese Formalisierung arbeitet mit denselben definierten Begriffen, hat also dieselbe Struktur. Dies spricht für die Objektivität der folgenden Beschreibung.

3.6.1 Die neue Beschreibung

Wir beginnen mit den Bestimmungen einiger elementarer Beobachtungen. Die Karte ist eingeteilt in verschiedene *Räume*, diese sind entweder ein *Meer* oder eine *Provinz* und auf der Karte mit ihrem Namen beschriftet. Zwei Räume können aneinander *angrenzen*; Provinzen, die an Meere grenzen, heißen *Küstenprovinzen*, alle anderen *Inlandprovinzen*. Küstenprovinzen können mehrere *Küsten* haben, diese sind dann ebenfalls mit einem Namen bezeichnet und sie grenzen an andere Räume und Küsten. All dies ist auf der Karte leicht zu erkennen, nötigenfalls geben andere Dokumente genauere Hinweise. Eine *Einheit* ist immer eine *Armee* oder eine *Flotte*. Jeder sieht den Unterschied, und an der Farbe erkennt er, zu welcher *Macht* die Einheit gehört. Armeen können nur Provinzen *besetzen* und Flotten nur Meere und Küstenprovinzen. In einer *Spielstellung* ist jeder Raum höchstens von einer Einheit besetzt. Eine Flotte in einer Provinz mit mehreren Küsten steht auf genau einer dieser.

Wir betrachten nur den Teil eines Spielzugs genannt “Angriffsphase”. In dieser Phase ist eine Spielstellung gegeben und eine Menge von Zügen der verschiedenen Spieler. Zunächst werden wir genauer bestimmen, was diese Züge sind. Dann geben wir mit Hilfe einiger Definitionen den Ausgang dieser Phase an.

Spielzüge: Jede Macht bestimmt für jede ihrer Einheiten auf der Karte einen *Zug*. Ein *Zug* ist entweder Angriff, Halten, Unterstützen oder Geleiten. Mit allen Zügen ist ein *Ursprungsraum* angegeben, dadurch wird die ziehende Einheit bezeichnet. Steht die ziehende Einheit auf einer Küste, so ist dies die *Ursprungsküste*. Mit *Angriffen* (synonym: *Bewegung*) ist ein *Zielraum* angegeben, nötigenfalls (d.h. wenn eine Flotte in eine Provinz mit mehreren Küsten zieht) eine *Zielküste*. Der Ursprungsraum (und die Ursprungsküste, falls vorhanden) grenzen an den Zielraum (und die Zielküste, falls vorhanden), und die Einheit kann den Zielraum besetzen. Ein Angriff durch eine Armee kann *per Geleitzug* erfolgen, Ursprungs- und Zielraum sind dann Küstenprovinzen. Mit Unterstützen und Geleiten ist ein *Zielzug* angegeben, dieser wird angegeben wie ein Zug, aber ohne Zielküste. Beim *Unterstützen* ist dies ein Geleiten, Halten oder ein Angriff und der Ursprungsraum des Geleitens/Haltens bzw. der Zielraum des Angriffs (ohne Beachtung der Küste) muss (a) an den Ursprungsraum der unterstützenden Einheit (und deren Küste, falls vorhanden) angrenzen und (b) von der unterstützenden Einheit besetzt werden können. Nur Flotten können *geleiten* und auch nur, wenn sie ein Meer besetzen. Der Zielzug ist immer ein Angriff.

Wir regeln hier nicht, ob Zielzüge auch existieren müssen (d.h. eine Einheit macht genau diesen Zug), dies kommt erst später in Betracht.

An das *Halten* stellen wir keine weiteren Bedingungen. Alle Züge, die nicht diesen Bestimmungen entsprechen, werden als “Halten” interpretiert. Ebenfalls “halten” alle Einheiten, die mehr oder weniger als einen Zug vorliegen haben.

Es folgt nun eine Reihe von Definitionen und am Schluss eine Aussage über *vorrückende Angriffe*. Wenn man diese kennt, ergibt sich eine neue Spielstellung ganz einfach, indem man die entsprechenden Einheiten versetzt und alle anderen stehen lässt (oder vertreibt, siehe Bemerkung am Schluss).

1. Ein Geleitzug ist die Menge aller Flotten, die denselben Angriff geleiten. Ein Geleitzug ist *gültig*, wenn die Ursprungsräume seiner Flotten einen Pfad vom Ursprungsraum zum Zielraum des Zielzuges bilden. Ein Angriff ist *gültig*, falls er ohne Geleitzug geschieht oder sein Geleitzug gültig ist.
2. Ein Geleitzug ist *erfolgreich*, wenn die Ursprungsräume seiner nicht vertriebenen Flotten (siehe letzter Punkt) immer noch einen Pfad vom Ursprungsraum zum Zielraum des Zielzuges bilden. Ein Angriff ist *wirksam*, falls er ohne Geleitzug geschieht oder sein Geleitzug erfolgreich ist.
3. Eine Unterstützung ist *abgeschnitten*, falls ihr Ursprungsraum Ziel eines gültigen Angriffs ist. Falls der unterstützte Zug ein Angriff auf den Raum ist, aus dem der abschneidende Angriff kommt, so sprechen wir von einer *Frontalunterstützung*. (Sie wird aber trotzdem abgeschnitten.)
4. Die *Angriffsstärke* und *Verteidigungsstärke* eines Angriffs bzw. Haltens ist gleich der Anzahl der nicht abgeschnittenen Unterstützungszüge für genau diesen Zug. Die Verteidigungsstärke von angreifenden und unterstützenden Einheiten (die selbst keine Unterstützung empfangen) ist 0 (Null). Die *Vertreibungsstärke* eines Angreifers ist wie seine Angriffsstärke, wir zählen aber (a) auch Frontalunterstützung und (b) keine Unterstützung von Einheiten derselben Macht, wie die Einheit im Zielraum des Angriffs.

Die offiziellen Diplomacy-Regeln enthalten eine Bestimmung (V), dass “eine Unterstützung immer wirkungslos ist, wenn die unterstützende Einheit vertrieben wird.” Diese Regel wird impliziert von (Z) “eine Unterstützung ist immer wirkungslos, falls die unterstützende Einheit Ziel eines gültigen Angriffs ist.” (Das Wort “wirksam” statt “gültig” würde auch reichen.) Die offiziellen Diplomacy-Regeln machen aber von (Z) eine Ausnahme bei der Frontalunterstützung und deswegen muss (V) manchmal prioritär angewendet werden. Das führt zu der Spezifikation: “Frontalunterstützung wird nicht abgeschnitten, es sei denn die unterstützende Einheit wird vertrieben.”

Auch unsere Regeln machen eine Ausnahme von (Z) (nämlich Punkt (a) eben), aber trotzdem wird die Bedingung (V) noch von unserer Spezifikation impliziert und muss nicht explizit genannt werden. Zum Beweis der Implikation betrachte man drei Räume A, B, C und Einheiten darauf, so dass die folgenden Züge möglich sind:

1 A → B

2 B → C

3 C unterstützt 1

Weiterhin nehmen wir an, dass C von keinem anderen Raum aus angegriffen wird, so dass es sich um Frontal-Unterstützung handelt. Wir wollen zeigen, dass Cs Unterstützung wirkungslos ist, falls C vertrieben wird. Obiger Punkt (a) spezifiziert, dass Cs Unterstützung in As Vertreibungsstärke “mitgezählt” wird. Diese Vertreibungsstärke ist aber nur wirksam, falls B nicht vorrückt (andernfalls hätte A niemanden zu vertreiben). Da B aber der einzige Angreifer gegen C ist, bedeutet sein Nicht-Vorrücken, dass C nicht vertrieben wird. Regel (V) kann also niemals verletzt werden.

Formalisten würden sagen, dass wir durch Herausarbeiten dieser Implikation die Regeln besser verstanden haben. Praktisch gesehen haben wir einerseits den Vorteil, die Regeln besser mit anderen vergleichen zu können: Welche Unterschiede sind nur Formulierung,

welche sind inhaltlich? Und wir haben mehr Freiheit bei der Anwendung der Regeln (manuell oder in einem Programm), weil wir implizierte Aussagen verwenden können, als wären sie Regeln, aber wir müssen sie nicht jedes Mal explizit betrachten, weil die anderen Regeln schon ausreichen.

Warum die hier gegebenen Regeln von den offiziellen abweichen, ist an anderer Stelle beschrieben [42]. Kurzform: wir stellen uns vor, dass das Zurückschlagen von Angriffen und das Vertreiben von Einheiten zeitlich hintereinander geschehen. In der ersten Phase (wenn A gegen andere Angreifer auf B kämpft) ist C offensichtlich “nicht verfügbar”, da von Bs Angriff abgelenkt. In der zweiten Phase ist C verfügbar, da sie sich bereits “in Front auf B” befindet. Die offiziellen Regeln spezifizieren, dass C (während sie “mit B beschäftigt” ist) auf Schlachten Einfluss nimmt, die “hinter B” stattfinden. Dieser anschauliche Widerspruch kommt dadurch zu Stande, dass die offiziellen Regeln versuchen, das Ergebnis aller Schlachten in einem Schritt anzugeben. Sie haben eben keine interne Struktur.

5. Ein wirksamer Angriff ist *zurückgeschlagen*, falls
 - a) Es gibt mindestens einen anderen wirksamen Angreifer mit demselben Zielraum und mindestens derselben Angriffsstärke.
 - b) Oder es gibt einen Angreifer mit umgekehrtem Ursprungs- und Zielraum und mindestens derselben Angriffsstärke. (Wir nennen das *Frontalschlacht*.)
6. Ein nicht zurückgeschlagener Angriff *rückt vor* (endlich!), falls
 - a) Sein Zielraum ist unbesetzt.
 - b) Oder der Zielraum wird von einer Einheit einer fremden Macht (d.h. anders als der Angreifer) besetzt, die selbst nicht vorrückt und deren Verteidigungsstärke kleiner als des Angreifers Vertreibungsstärke ist.

Diese Regel ist rekursiv und dadurch unvollständig. Von allen möglichen Lösungen ist diese zu wählen, welche maximal viele Angreifer vorrücken lässt.

Hier haben wir den kleinsten Fixpunkt ausgewählt, welcher übrigens immer existiert, da die Regel monoton ist: das Vorrücken einer Einheit lässt andere nachrücken und behindert nie das Vorrücken einer anderen Einheit. (Dies wurde durch das Zurückschlagen schon erledigt.) Etwas formaler würden wir in der Regel erst “vorläufiges Vorrücken” definieren und dann Vorrücken als kleinsten Fixpunkt des vorläufigen Vorrückens, nachdem wir natürlich noch definiert haben, dass Lösungen kleiner sind, wenn mehr Einheiten vorrücken. (Fixpunkte besprechen wir kurz in Abschnitt 4.1.3.)

Übrigens kann man so auch den Entscheidungsprozess durchführen: man lässt Einheiten auf der Karte vorläufig vorrücken, wenn sie nur von einer anderen Einheit nicht zurückgeschlagenen Einheit behindert werden. Nur wenn diese Einheit dann nicht vorrückt, kann die vorläufige Einheit auch nicht vorrücken.

7. *Vertriebene Einheiten* sind all diejenigen, die den Zielraum eines erfolgreichen Angriffs besetzen und selbst keinen erfolgreichen Angriff durchgeführt haben.

Eine weitere Rekursion entsteht durch die Bezugnahme auf die vertriebenen Flotten in der Definition eines erfolgreichen Geleitzuges, welche dann indirekt über fast alle anderen Definitionen in der Definition der vertriebenen Einheiten vorkommen. Man kann sich folgendermaßen überlegen, dass dadurch keine Zyklen entstehen können. Der Erfolg eines Geleitzuges hat eine Auswirkung an seinem Ursprungs- und seinem Zielraum. Am Ursprung geht es darum, ob andere Einheiten nachrücken können, dies kann keinen Einfluss auf einen Geleitzug nehmen, da dieser nur von Einheiten beeinflusst wird, die auf vom Geleitzug besetzte Felder vorrücken (wollen). Am Ziel geht es einerseits um abgeschnittene Unterstützung, diese ist aber unabhängig vom Erfolg des Geleitzuges, und andererseits um das Zurückschlagen von Angriffen und Vertreiben von Einheiten. Beide können keinen Einfluss haben, weil das Ziel eine Provinz ist und nur Flotten auf See an einem Geleitzug teilnehmen können.

Diese Überlegung hilft wieder beim Entscheidungsprozess: man löse einfach zuerst alle Schlachten auf Meeren, bestimme dann erfolgreiche Geleitzüge und löse dann die Schlachten in Provinzen.

Was mit den vertriebenen Einheiten passiert, wird in der “Rückzugsphase” geklärt, die sich auch leicht beschreiben lässt [49], was für unsere Zwecke aber nicht nötig ist. Wir haben den schwierigsten Teil der Diplomacy-Entscheidung gesehen. Geschickt beschrieben ist er gar nicht so kompliziert, oder?

4 Formale Theorien der Programmierung

Eine Formale Theorie der Programmierung beschränkt sich komplett auf deren formale und mathematische Aspekte. Mathematisch ist dabei alles, was sich im Schema von Definition, Satz, Beweis abhandeln lässt; insbesondere wenn dabei weitere aus der Mathematik bekannte Konzepte eine Rolle spielen: Funktionen, Mengen, Abgeschlossenheit usw. Und “formal” bedeutet, dass man abstrakte Objekte nur nach festen Regeln untersucht, ohne sich dabei hauptsächlich um die Bedeutung zu kümmern. Die Ansprüche solcher Theorien gehen in zwei Richtungen: Einerseits nach den grundlegenden mathematischen Theoremen der Programmierung und allgemeingültigen formalen Eigenschaften von Programmiersprachen: was bedeuten Programme? Was haben verschiedene Paradigmen gemeinsam; wie hängen sie zusammen? Andererseits möchte man sich formale Eigenschaften für die Praxis zu Nutze machen: Wie kann man die Programmentwicklung und -verifikation mathematisch unterstützen? Wie kann man Programmverhalten praktisch abstrakt beschreiben (also spezifizieren), und wie kann man aus einer solchen Spezifikation (möglichst ohne viel Kreativität) ein Programm herleiten?

Wir haben hier also eine philosophisch-theoretische Seite und eine methodische Seite. Diese Arbeit interessiert sich natürlich mehr für die Methodik, aber die Theorien lassen sich am besten verstehen, wenn man beide Aspekte gemeinsam betrachtet. Die “theoretischen Theorien” der Programmierung kann man ungefähr so einordnen wie die Theorie der Berechenbarkeit für die Informatik allgemein. Sie sind aber nicht von so allgemeiner Bedeutung, weil man ihre Ergebnisse (die Hauptsätze sozusagen) als einfacher Informatiker nicht unbedingt einordnen kann. Für die Entwicklung von Programmiersprachen sind die Theorien aber angeblich durchaus von Bedeutung — besonders der Begriff der Monotonie.

In diesem Kapitel möchte ich zunächst grob alle wichtigen Theorien benennen und einordnen. Ich orientiere mich dabei an den fünf Stichwörtern: denotational, axiomatisch, prädikativ, algebraisch und operational. Danach gehe ich kurz auf die modernste der “praktischen Theorien” ein. Diese wurde speziell dazu entworfen praktisch zu sein: Programmierprobleme mit formaler Hilfe lösen und deren Lösungen verifizieren.

Wenn ich nicht doch eine wichtige Publikation verpasst habe, so ist der Überblick in diesem Kapitel im deutschsprachigen Raum einzigartig — scheinbar hat noch niemand so einen Vergleich angestellt. Überhaupt sind alle Bücher, die ich zum Thema kenne, nur auf Englisch verfügbar, aber auch in dieser schönen Sprache ist mir kein solcher Überblick bekannt.

4.1 Alte und neue Theorien

Ich möchte hier ein etwas vereinfachtes Bild präsentieren, dass einige Zusammenhänge leichter klar werden lässt: die denotationale und axiomatische Semantik sind die alten Theorien, die prädikative und algebraische Semantik sind die neuen Theorien, und die operationale Semantik war schon immer da.

4.1.1 Operationale Semantik

Die operationale Semantik ist historisch die erste; sie entsprang dem Drang genauer zu beschreiben, was dieses und jenes neue Konstrukt einer Programmiersprache denn eigentlich genau macht. Das war in einer Anfangszeit, in der immer mehr Programmiersprachen entwickelt wurden, aber die Methoden dazu (wenn man überhaupt von “Methoden” reden kann) noch in den Kinderschuhen steckten. Bis in die 50er Jahre gab es noch nicht einmal formale Grammatiken zur Syntax-Beschreibung, und als einzige semantische Referenz hatten die Programmierer ih-

re Computer, deren Maschinsprache sie aus dem Effeff kannten. Nur leider gab es damals für jeden Computer eine neue, verschiedene Maschinsprache, und man möchte ja, dass eine “höhere” Programmiersprache von diesen technischen Details unabhängig ist. Da lag es doch nahe, sich einfach eine abstrakte Maschine zu definieren und dann jedes einzelne Konstrukt einer Programmiersprache darauf abzubilden! Die operationale Semantik erklärt also die Bedeutung von Programm-Konstrukten durch deren Wirkung auf eine abstrakte Maschine.

Diese abstrakten Computer kann man sich vorstellen wie die Register-Maschinen aus der theoretischen Informatik. Man hat die Semantik einer Programmiersprache einfach definiert, indem man sie (grob gesagt) in Flussdiagramme für solche Register-Maschinen übersetzte! Auf diesem primitiven Niveau wurde damals jede Programmvariable direkt einer Speicherzelle des Computers zugeordnet. Später kam man auf die Idee, sich den Speicher einer abstrakten Maschine als Stapel vorzustellen, und das war ein großer Durchbruch: man konnte dadurch prima lokale Variablen implementieren und auch spezifizieren, Register wurden unnötig, die Beschreibung und Implementierung von Programmiersprachen wurde einfacher, und plötzlich waren auch rekursive Prozeduren möglich. Die Sprache Algol’60 machte in ihrer Entwicklung intensiv von der Idee der Stapel Gebrauch und wurde dadurch sehr einfach und trotzdem ausdrucksstark. Ein erster Erfolg der Theorie in der Praxis!

Auch heute noch spielt die operationale Semantik für Compiler-Bauer eine Rolle, in vielerlei Hinsicht ist sie aber überholt: wenn man in den Anleitungen moderner Programmiersprachen immer noch Referenzen auf Konzepte wie *stack* und *heap* findet, so ist das ein Mangel von Abstraktion: diese Sprachen sind auf dem Niveau der 70er Jahre stecken geblieben.

In der Tat kommen die meisten Programmiersprachen nicht über die 80er Jahre hinaus, denn damals wurden Generizität und DESIGN BY CONTRACT entwickelt, die den meisten Sprachen heute noch fehlen.

[21] bezeichnet die operationale Semantik auch als nützlich zum Verstehen der Programmausführung und die Interpretation von Debugging-Ausgaben, wie zum Beispiel Stapel-Spuren (*stack-traces*). Ich meine, dass man mit Laufzeittests von Zusicherungen (auf axiomatischer Basis) viel besser debuggen kann. Gute Laufzeitsysteme versöhnen die beiden Ansichten, indem sie bei jeder fehlgeschlagenen Zusicherung nicht nur die Code-Zeile und die Variablen-Belegung ausgeben, sondern auch den Stapel aller aktiven Routinen plus deren lokale Variablenbelegung.

Mehr Informationen zur operationalen Semantik sowie formale Relationen zu den anderen Semantiken findet der Leser in [30, 21].

4.1.2 Axiomatische Semantik

Wie wir später noch sehen werden, ist die axiomatische Semantik heute noch von viel größerer praktischer Bedeutung. Sie geht auf Methoden der Programmverifizierung zurück, die erstmalig von Floyd [6] veröffentlicht wurden, auch vom berühmten Knuth so verwendet (die beiden waren damals zusammen in Stanford die Informatik-Gurus), aber auch unabhängig von Peter Naur — er nannte die Zusicherungen (*assertions*) allerdings *snapshots*. Ein großer Moment für die Informatik: 1969 bekam Tony Hoare Floyds Artikel in die Finger und übertrug die Methode von Flussdiagrammen auf strukturierte Programmierung. Der resultierende Artikel [18] ist Legende. In einem Folgeartikel [19] erweitert Hoare das Konzept auf imperative abstrakte Datentypen (ADTs) und legt damit in gewisser Weise den Grundstein für die Objektorientierung. Und das 1972! Beide Artikel sind übrigens heute noch aktuell und auch für Studenten lesbar. Meine Empfehlung!

Tony Hoare hat die Beweismethode damals an der Sprache SIMULA 67 vorgeführt, der ersten Sprache mit dem Konzept einer “Klasse” wie wir es heute kennen. Hoare hat also nicht das programmiersprachliche Konstrukt erfunden (diese Idee kommt Ole-Johan Dahl und Krysten Nygaard zu), sondern nur die Beweisregel dazu. Und genau das war der wichtige Schritt, denn Konstrukte zur Aufteilung von Programmtexten gibt es viele, aber auch das *Verständnis* über ein Programm aufzuteilen ist die wahre Kunst, die die Programmierung vereinfacht.

Übrigens sprach man damals noch gar nicht von Objektorientierung, diesen Namen hat erst Alan Kay mit seiner Sprache SMALLTALK eingeführt, die auch von SIMULA inspiriert war.

Die grundsätzliche Idee der axiomatischen Semantik besteht darin, den Zustand von Variablen vor und nach einer Berechnung durch Prädikate zu beschreiben. Hier haben wir also erstmals eine zweite *formale Sprache* neben der Programmiersprache. In dieser Sprache drückt man *Spezifikationen* aus, die aus einer *Vorbedingung* und einer *Nachbedingung* bestehen. Der sogenannte Hoare-Kalkül besteht aus zwei Teilen (ganz im Sinne der Leibnitz'schen *lingua characterista* und *calculus ratorator*): Aussagen der Form $\{pre\}P\{post\}$ (den sogenannten Hoare-Tripeln), und Beweisregeln mit denen man solche Aussagen herleiten kann. Die geklammerten Teile des Hoare-Tripels sind die Vor- und Nachbedingung, und P ist das Programm. Beweisregeln gibt es für jedes elementare Konstrukt der strukturierten Programmierung: Zuweisung, Sequenz, Fallunterscheidung, Schleife und das leere Programm. (Mit der Axiomatisierung von Routinen und den verschiedenen Arten der damals üblichen Parameterübergabe hat man sich damals allerdings schwer getan; wir werden später noch sehen wie die Verfeinerung eine einfache Antwort liefert.) Die Regel für die Fallunterscheidung ist zum Beispiel die folgende: Gegeben die Aussagen $\{pre \wedge b\}P\{post\}$ und $\{pre \wedge \neg b\}Q\{post\}$ über Teilprogramme, folgern wir $\{pre\}if\ b\ then\ P\ else\ Q\ end\{post\}$.

Man sieht daran schon, wie die Hoare-Semantik zur schrittweisen Programmverifikation eingesetzt werden kann, und das ist ihre geniale Eigenschaft: sie ist *gleichzeitig* eine formale Definition der (strukturierten) Programmiersprache *und* eine praktische Methode zur Programmverifikation. Wie wir noch sehen werden, liegt aber darin auch der Nachteil der Theorie: man kann damit die Korrektheit eines Programms nur *überprüfen*, aber nicht von vornherein *entwickeln*. Aus diesem Aspekt heraus entwickeln sich die Theorien der Verfeinerung, zu denen wir gleich noch kommen.

Seinen größten Einfluss auf die Praxis der Programmierung hat der Hoare-Kalkül aber durch die *Zusicherungen* erreicht. Diese Aussagen, die man leicht innerhalb der Programmiersprache selbst formulieren kann, sind von unglaublichem praktischen Nutzen.

1. Als Vor- und Nachbedingungen sind sie *die* Spezifikationsmethode der imperativen Programmierung.
2. Als Invarianten formalisieren sie die algorithmische Idee von Schleifen und von Implementierung abstrakter Datenstrukturen.
3. Als Zusicherungen zwischen Anweisungen erlauben sie gezieltes Nachvollziehen "logischer Berechnungen".

Zusicherungen dienen gleichzeitig der Dokumentation und dem automatischen Test — besser kann sich eine formale Methode kaum auf die Programmierung auswirken.

4.1.3 Denotationale Semantik

Nachdem wir jetzt zwei Theorien kennen gelernt haben, die aus der praktischen Notwendigkeit heraus entstanden sind, wenden wir uns hier erstmals der Frage zu: "Was *ist* eigentlich ein Programm?" Die Antwort der denotationalen Semantik lautet: "Programme bezeichnen (engl. *to denote*) mathematische Funktionen." Die Wissenschaftler Dana Scott und Christopher Strachey haben für ihre Arbeiten zu dieser Frage den Turing-Award bekommen, und in seiner Preisvorlesung [37] hat Scott ein Stück der Theorie vorgestellt. Das ist aber schon so abstrakt, dass man kaum noch etwas von der Programmierung darin wiedererkennt. In diesem Artikel reduziert Scott alle Berechnungen auf zwei Klassen: terminierende und nicht-terminierende. Er sieht Berechnungen, die nicht terminieren (oder das auf weniger Eingaben tun) als *kleiner* an als (öfter) terminierende Berechnungen und kommt dadurch zur Monotonie: offensichtlich kann kein implementierbarer Operator zur Kombination von Berechnungen nicht-terminierende Berechnungen

in terminierende Umwandeln. Solche Operatoren können Berechnungen also immer höchstens kleiner machen — sie sind monotone Funktionen.

Bei Bertrand Meyer [30] findet man eine leicht verständliche Version dieser Theorie, deren Bezug zur Programmierung wieder klar erkennbar ist. (Meyer schreibt auch, dass er nur einen Spezialfall von Scotts Theorie präsentiert, weil sie im Ganzen zu allgemein ist...) Aus diesem Buch habe ich das wichtige Konzept vom *kleinsten Fix-Punkt* gelernt. Mit kleinsten Fix-Punkten kann man Rekursion “formal verstehen”, die ja in formalen Sprachen so oft vorkommt. Schauen wir uns dazu das kanonische Standard-Beispiel an:

$$fak(n) \equiv \text{if } n = 0 \text{ then } 1 \text{ else } n * fak(n - 1) \text{ end}$$

Was ist wohl die Semantik dieser rekursiven Definition? Als Programmier-Anfänger wird man sagen: Na, wenn man die Funktion mit n aufruft, dann kommt $n * fak(n - 1)$ heraus, also noch ein Aufruf; das ergibt dann $n * (n - 1) * fak(n - 2)$ und so weiter bis n schließlich 0 ist (Basisfall). Ergebnis ist also $fak(n) = n * (n - 1) * (n - 2) \cdots 1 = n!$. Das ist eine typisch operationale Denkweise. Damit kommt man in der Praxis nicht besonders weit (höchstens, um sich Ideen und Hypothesen zu verschaffen), und in der Theorie schon gar nicht.

Wenn man schon professioneller programmiert, wird man die Hypothese “ $n \geq 0 \Rightarrow fak(n) = n!$ und die Funktion terminiert nach n Selbstaufrufen” leicht axiomatisch verifizieren: im Fall $n = 0$ ist das Resultat $1 = 0!$ ohne Selbstaufruf; im Fall $n > 0$ ist das Resultat $n * fak(n - 1)$, da $n - 1 \geq 0$ können wir die Hypothese anwenden und erhalten $n * (n - 1)! = n!$ mit $1 + (n - 1) = n$ Selbstaufrufen und das war’s. Man beachte, dass wir den Terminierungsbeweis brauchen, auch wenn wir nur die “Nachbedingung” wissen wollen. Sonst könnten wir nämlich auch irgendwelche Sachen für negative Argument-Werte beweisen, für die die Funktion aber niemals etwas zurückliefert. (Außer einen Stack-Overflow, operational gesehen. Dieses Problem erläutere ich näher unter der Prädikativen Semantik. Außerdem können wir natürlich mit der axiomatischen Methode vielerlei Dinge über fak beweisen (z.B. $fak(3) = 6$, $n > 10 \Rightarrow fak(n) = n!$ oder $n \geq 2 \Rightarrow$ “ $fak(n)$ ist gerade”), aber wir erhalten nicht die eindeutige Aussage: was ist die absolute alleingültige Semantik von fak ?

Die Fixpunkt-Theorie sagt dazu Folgendes: wir interpretieren obige Definition von fak als eine Gleichung, die die freie Variable fak einschränkt. Erste Bemerkung: die Gleichung hat die Form $fak = F(fak)$, wobei F der Ausdruck auf der rechten Seite ist, mit fak als Parameter, also $F = \lambda fak, n. \text{if } n = 0 \text{ then } 1 \text{ else } n * fak(n - 1) \text{ end}$. Funktionen fak , die diese Gleichung erfüllen, sind also Fixpunkte von F . Daher der Name. Zweite Bemerkung: Aussagen, die wir wie eben axiomatisch beweisen, gelten für alle Funktionen, die die Gleichung erfüllen. Wir könnten uns nun entscheiden, dass unsere Funktion fak diese Menge der die Gleichung erfüllenden Funktionen bezeichnet. Wir wollen aber, dass fak genau eine Funktionen bezeichnet, also müssen wir aus dieser Menge der Fixpunkte einen herausuchen. Hier erinnern wir uns, dass wir auf der Menge der Funktionen eine Ordnungsrelation definiert haben. (Eine nicht-terminierende Berechnung, von der wir eben sprachen, entspricht dabei einer nicht-definierten Funktion.) Außerdem kommt in der Theorie noch vor, dass diese Ordnungsrelation ein Gitter ist (engl. *lattice*), dass also zwei beliebige Funktionen nicht unbedingt zu ordnen sind (es muss nicht unbedingt eine von ihnen die kleinere sein, wie das bei einer Totalordnung wäre), aber dass zu zwei beliebigen Funktionen immer genau eine existiert, die kleiner ist als beide und mindestens so groß wie alle anderen, die kleiner sind als beide.

Kurz und gut, hier der Hauptsatz der Fixpunkt-Theorie: “Eine Gleichung der Form $f = F(f)$ hat genau dann einen kleinsten Fixpunkt (*least fix-point*), falls F monoton ist.”

Dies gesichert, sagen wir jetzt einfach: eine rekursive Definition bezeichnet genau die Funktion, die ihr kleinster Fixpunkt ist. Dessen eindeutige Existenz ist gesichert, weil alle Programmiersprachkonstrukte monoton sind. Jetzt kennen wir einen weiteren Grund, warum das auch gut so ist. Ich habe übrigens diese Überlegung schon zweimal im (praktischen) Leben gebraucht. Das erste Beispiel habe ich ja bereits im letzten Kapitel vorgestellt, das zweite ist “Generic Conformance” und Informationen dazu finden sich auf derselben Webseite.

Peter Pragmatiker hat mich gerade daran erinnert, dass man eigentlich gar keine denotationale Semantik braucht, um eine eindeutige Semantik eines Programms zu definieren. Man

kann sich ja auch ganz einfach die Implikations-Relation auf den Prädikaten der axiomatischen Semantik nehmen und dann definieren, dass die Semantik eines Programms die stärkste Spezifikation ist, die noch vom Programm erfüllt wird. Leider klappt das aber nicht, weil die Spezifikationen ja aus *zwei* Prädikaten bestehen.

Edsger Dijkstra hat dafür eine Lösung gefunden: der Kalkül der kleinsten Vorbedingung. Er bildet Programme auch auf Funktionen ab, aber nicht Funktionen über Werten, sondern Funktionen über Prädikaten: ein Programm ist eine Funktion, die eine Vorbedingung in eine Nachbedingung umwandelt. Dadurch kann er so praktisch rechnen, wie mit Spezifikationen und Zusicherungen, hat aber den (rein theoretischen!) Vorteil einer eindeutigen “Bedeutung” jedes Programms! (Dass dies so ist, wurde mir auch eben erst klar. Es ist für die eigene Wissensverarbeitung doch gar nicht so schlecht, ab und zu 'mal eine Diplomarbeit zu schreiben.)

4.1.4 Algebraische Semantik

Jetzt kommen wir langsam ins neuzzeitliche Lager. Diesmal ist unser Ansatzpunkt, dass die ansonsten doch sehr praktische axiomatische Semantik von der Prädikatenlogik abhängt und in der ganz formalen Version auch von einem System logischer Schlussregeln (viele Autoren, darunter auch Gries [11] verwenden Gentzens “Natürliches Schließen”), das ich dem Leser hier erspart habe, weil wir gleich etwas besseres kennen lernen. Die Idee des algebraischen Ansatzes ist, Spezifikationen derart *abgeschlossen* (*self-contained*) aufzuschreiben, dass sie alle Informationen enthalten, um daraus Schlussfolgerungen ziehen zu können — ohne zusätzliche Logik. Dies zumindest ist das Ziel, das heute noch gültig und von großer Bedeutung ist. Leider hat sich in der Literatur eine viel restriktivere Sicht der Dinge durchgesetzt, so dass ich mich hier berufen fühle, diesen Widerspruch mal aufzuklären.

Die “Urpublikation” zu algebraischen Spezifikationen stammt von Joseph Goguen und Rod Burstall [4] und wurde interessanterweise auf einem Kongress zur künstlichen Intelligenz veröffentlicht! Die Autoren ersetzen dort die übliche formale Beschreibungsweise mit Prädikaten und Werten durch eine reine Welt von Werten, insbesondere Funktionen. Sie geben Spezifikationen in Form von “Theorien” an, die im Wesentlichen aus Gleichungen bestehen. Dadurch kann man direkt Schlussfolgerungen aus der Spezifikation ziehen und ein formales System aufbauen, indem man eine Implementierung konstruiert. Selbst die größten Wissenschaftler verwenden immer wieder Stapel als Beispiel für eine solche algebraische Spezifikation. Ich will wieder mal cooler sein als die anderen und verwende Sequenzen mit den primitiven Funktionen *empty*, *single*, *++*, *is_empty*, *first*, *reverse* und den folgenden Axiomen:

$$\begin{aligned} & is_empty(empty) \\ & \forall x \cdot \neg is_empty(single(x)) \\ & \forall a, b \cdot is_empty(a ++ b) \equiv is_empty(a) \wedge is_empty(b) \end{aligned}$$

$$\begin{aligned} & \forall a \cdot empty ++ a \equiv a \\ & \forall a \cdot a ++ empty \equiv a \\ & \forall a, b, c \cdot a ++ (b ++ c) \equiv (a ++ b) ++ c \end{aligned}$$

$$\begin{aligned} & \forall a, b \cdot a \neq empty \Rightarrow first(a ++ b) \equiv first(a) \\ & \forall x \cdot first(single(x)) \equiv x \end{aligned}$$

$$\begin{aligned} & \forall x, a \cdot reverse(single(x) ++ a) \\ & reverse(empty) \equiv empty \end{aligned}$$

Einige Folgerungen aus diesen Axiomen sind zum Beispiel:

$$\begin{aligned} & \forall a \cdot reverse(reverse(a)) \equiv a \\ & \forall x, a \cdot first(single(x) ++ a) \equiv x \end{aligned}$$

Gleich werden wir sehen, wie man auf ähnliche Weise die logischen Operatoren spezifizieren kann und damit auf Folgerungssysteme höherer Ordnung verzichten kann. Aber erst einmal muss Goguens genialer Gedanke noch seinen Weg in die Theorie der Programmierung finden.

Richtig bekannt wurde der algebraische Ansatz, als man ihn zur Spezifikation von abstrakten Datentypen verwendete. John Guttag wurde berühmt, weil er in seiner Doktorarbeit gezeigt hat, dass eine algebraische Spezifikation in einer bestimmten Normalform vollständig und widerspruchsfrei ist. Aus pragmatischer Sicht ist das aber ein triviales Resultat, weil diese Normalform genau einem Funktionalen Programm mit Pattern-Matching entspricht — man kann also kaum noch von Spezifikationen sprechen! Seit dieser Zeit ist aber der Begriff des abstrakten Datentyps im Bewusstsein der Informatiker unzertrennlich mit der algebraischen Spezifikation verbunden. Dabei trennt die algebraische Spezifikation und die objektorientierte Programmierung eigentlich ein unauflösbarer Widerspruch: die mathematischen Algebren beschreiben nur *Werte*, also unveränderliche Objekte; in der Programmierung haben wir es aber im Allgemeinen Fall mit *veränderlichen* Objekten zu tun, nämlich mit “kleinen Maschinen” (Bertrand Meyer), abstrakten Maschinen (J.-R. Abrial), oder instanziiierbaren Modulen, wie auch immer man sie nennen möchte, im Kapitel über Programmbausteine werden wir sie uns näher ansehen. Algebraische Spezifikationen haben ihr natürliches Anwendungsgebiet also eigentlich in der (puren) funktionalen Programmierung, während man Objekte wie Module spezifizieren sollte: mit Vor- und Nachbedingungen aller Routinen. Hier sollte eigentlich der Artikel [19] den Ansatzpunkt geben, aber anscheinend hat sich die Wissenschaft auf den algebraischen Ansatz fixiert und ist eher bereit ihn durch Konstruktionen wie Coalgebren zu reparieren als ihn aufzugeben. Der aufmerksame Leser kann sich vielleicht schon denken, wie wir die Algebren und Objekte noch versöhnen, aber erst einmal wollen wir ja Goguens genialen Gedanken weiterverfolgen...

Die gute Seite der Geschichte der Algebren geht wie folgt weiter: Edsger Dijkstra war der Meinung, dass man Korrektheitsbeweise in seinem Kalkül eigentlich mit Computern automatisch überprüfen können sollte ([5], obwohl diese Prophezeiung auch schon von anderen gemacht wurde) und widmete den Rest seines Lebens der Formalisierung von Programmentwicklungsprozessen. Zusammen mit seinem Kollegen Scholten entwickelte er ein logisches Schlussystem, das nur noch drei Folgerungsregeln enthielt — im Gegensatz zu den zwölf Grundregeln des natürlichen Schließens, denen man im Laufe der Anwendung meist noch einige Hilfsregeln hinzufügte. Und diese drei goldenen Regeln regelten nicht weniger als die Anwendung (algebraischer) Axiome, insbesondere der Gleichheit! Deswegen hieß ihr System übrigens auch *equational logic*, was ich hier mal ganz untreu mit *algebraischer Logik* übersetze, denn ihre Besonderheit liegt gerade darin, dass die logischen Operatoren \wedge , \vee und \Rightarrow genauso behandelt werden, wie andere Operatoren auch.

David Gries erkannte dann auch gleich, dass sein erstes und viel gefeiertes Buch [11] eigentlich viel zu kompliziert ist und schrieb mit seinem Kollegen Fred B. Schneider ein neues [10], in dem die beiden die volle Schönheit der algebraischen Logik herausarbeiteten. An Stelle von Prädikaten und Schlussregeln, gibt es nur noch Werte und Gleichheit (ein algebraischer Operator wie jeder andere). Die bekannten Grundregeln der Logik definiert man einfach also Axiome der Bool’schen Werte und Operatoren! Es scheint so, als ob diese Methode vollständig Computerprüfbar und gleichzeitig sehr (Menschen-)verständliche Beweise produziert, denn man kann das System trotzdem noch mit lokalen Annahmen verbinden, wie man sie aus herkömmlichen Beweisen kennt [1].

Wir haben in der algebraischen Logik zum Beispiel die folgenden Axiome (T und F stehen für die beiden Wahrheitswerte):

$$\begin{aligned} \neg F &\equiv T && \text{(Definition von } \neg \text{ und } F.) \\ T \vee a &\equiv T \\ F \vee a &\equiv a \\ a \vee b &\equiv b \vee a && \text{(Drei Teile der Definition von } \vee .) \\ a \Rightarrow b &\equiv \neg a \vee b && \text{(Definition von } \Rightarrow .) \end{aligned}$$

Damit können wir die Folgerungsregel “Modus Ponens” beweisen, d.h. aus a und $a \Rightarrow b$ folgt b :

$$\begin{aligned}
& a \Rightarrow b && \text{(Da } a \text{ gegeben ist, ersetzen wir es durch } T \text{.)} \\
= & T \Rightarrow b && \text{(Definition von } \Rightarrow \text{.)} \\
= & \neg T \vee b && \text{(Definition von } \neg \text{.)} \\
= & F \vee b && \text{(Definition von } \vee \text{.)} \\
= & b &&
\end{aligned}$$

Im Hinblick auf das vorige Kapitel ist anzumerken, dass die Abschaffung der besonderen Stellung von Prädikaten nicht auf die Theorie der formalen Beschreibungen, unsere Phänomenologie, ausgeweitet werden darf. Nur wenn man rein formal über die mathematischen Werte innerhalb des Computers nachdenkt, kann man Prädikate und Bool'sche Ausdrücke gleichsetzen.

In Kennerkreisen hat sich algebraische Logik durchgesetzt; wir werden sie mit der praktischen Theorie im nächsten Abschnitt näher kennenlernen.

4.1.5 Prädikative Semantik

Zwei Probleme hat unser bisher viel gelobter Hoare-Kalkül: er hängt von einem logischen System ab — davon haben in die Algebren ja befreit — und er unterstützt Verifikation mehr als Konstruktion — auch dessen hat man sich früh in der Geschichte angenommen. Schon früh wurde festgestellt, dass man die Regeln des Hoare-Kalküls auch rückwärts anwenden kann: man beginnt mit einer Spezifikation und einem unbekanntem Programm als “Lücke”, dann setzt man für das Programm eines der Konstrukte ein (Fallunterscheidung, Sequenz, ...) und hat eine oder mehrere neue Lücken. Der Begriff der *Verfeinerung* wurde in einem historischen Artikel von Niklaus Wirth geprägt [51]. In den späten 80ern und in den 90er Jahren wurde diese Methode von verschiedenen Wissenschaftlern formalisiert und zu einem Verfeinerungskalkül weiterentwickelt. Ein gutes Buch darüber ist [34]. Diese Kalküle sind bereits sehr gute Hilfsmittel zur Programmentwicklung per Hand, sie erben aber dafür einige andere Nachteile des Hoare-Kalküls. Zum Beispiel ist die Semantik der Spezifikationen nur andeutungsweise gegeben und formal nur über die Beweisregeln für Programme definiert. Wir haben hier also das formale Beschreibungsproblem aus dem letzten Kapitel: die Spezifikation erhält erst durch Programme ihre Bedeutung. Das ganze wird noch erschwert durch ein paar Extremfälle von Programmen, wie zum Beispiel das “Wunder” oder “Wächter” (die teilweise “wundervoll” sind). Diese Extrema sind zwar formal wohldefiniert und beim Rechnen ganz nützlich (so in etwa wie imaginäre Zahlen), aber was sie eigentlich wirklich bedeuten, lässt sich schwer sagen.

Die theoretische Lösung zu diesen Problem ist wieder schön einfach. Sie stellt den “Spezifikationen sind Programme”-Ansatz auf eine einheitliche Basis: Spezifikationen und Programme sind Prädikate! Man braucht nur noch einen einzigen Bool'schen Ausdruck, der Vorher- und Nachherzustand (genannt die “Berechnung”) gleichzeitig einschränkt. Wir verwenden dabei für jede Programmvariable x zwei logische Variablen: x steht für den Wert vor der Programmausführung und x' steht für den Wert danach. Die Zuweisung $x := E$ hat dann die einfache Semantik $x' = E \wedge y' = y$ (für alle anderen Programm-Variablen y) und **if** b **then** P **else** Q **end** hat die Semantik $(b \wedge P) \vee (\neg b \wedge Q)$. Man kann dann leicht Spezifikationen an der Realität überprüfen: die Variablenbelegung vor und nach einer Berechnung muss dem Prädikat entsprechen. Vorbedingungen schreibt man als einfache Implikationen. Und der formale Zusammenhang von Spezifikationen und Programmen ergibt sich direkt aus der Definition: Programme müssen natürlich mindestens dieselben Berechnungen beschreiben wie die Spezifikation. Verfeinerung oder “Implementierung” ist also auch eine einfache Implikation! Das macht einiges einfacher!

Eric Hehner und Tony Hoare haben dafür den Begriff *Prädikative Programmierung* eingeführt, und Hoares Vereinigte Theorie der Programmierung [21] zeigt auf, wie erweiterbar dieses Konzept ist: wenn man noch mehr Aussagen in einer Spezifikation einschränken will, führt man einfach neue Variablen ein, zum Beispiel für Programmlaufzeit oder interaktive Kommunikationskanäle. Die Vereinigte Theorie ist wieder klar dem theoretischen Zweig der Theorien zuzuordnen. Die Autoren bezeichnen ihren Ansatz übrigens wieder als *denotational*, denn Programme werden wieder direkt mit ihrer Bedeutung gleichgesetzt, nur diesmal sind es Prädikate, keine

Funktionen. Die prädikative Semantik ist aber der Programmierung viel näher (und auch unserer praktischen Theorie im nächsten Abschnitt), deswegen möchte ich den theoretisch angehauchten Begriff “denotational” lieber vermeiden.

Die prädikative Semantik ist auch eine Art *explizite* Semantik: man macht nur Aussagen über das, was wirklich da steht. Wenn man über Terminierung nachdenken möchte, führt man ein Prädikat ein, das die elementare Beobachtung “Programm terminiert” repräsentiert. Dieses Vorgehen hat erstens den Vorteil der Modularität (oder Trennung der Angelegenheiten): man redet nur, worüber man wirklich will. Und zweitens kann man zur Verknüpfung die normalen logischen Operatoren verwenden: Spezifikationen (und Programme!) kann man verknüpfen wie die Beschreibungen aus dem vorherigen Kapitel, das ist praktisch und einfach zu verstehen. (Im Gegensatz zur der “nicht-deterministischen Auswahl”, die in manchen Verfeinerungskalküli an Stelle des Odors vorkommt.) Man kann sagen, dass die Vereinigte Theorie Konstrukte aus der Programmierung prädikativ definiert, um dann algebraische Eigenschaften dieser Konstrukte zu bestimmen!

Zu guter Letzt hilft uns die prädikative Semantik, eine noch offene Angelegenheit zu erklären: Ich hatte bereits bemerkt, dass uns die axiomatische Semantik alle möglichen Unwahrheiten über Programme beweisen lässt, wenn wir nicht gleichzeitig beweisen, dass das Programm auch terminiert. Ich hatte dabei verschwiegen, dass es vom Hoare-Kalkül sogenannte *partielle* und *totale* Versionen gibt: erste arbeitet ohne Terminierungsaussagen, letztere mit. In der Praxis ist die erste Version von sehr zweifelhaftem Nutzen, denn wenn man bei ihrer Anwendung nicht aufpasst, kann man leicht etwas Unnützes beweisen, weil das Programm gar nicht terminiert! Die Verfeinerungskalküli arbeiten also alle mit totaler Semantik, und deswegen sind sie auch komplizierter als die prädikative Semantik. Aber Hoare und He [21] zeigen uns, dass das Schöne und das Nützliche vereinbar sind: man kann in der prädikativen Semantik ein Prädikat t einführen mit der Bedeutung “das Programm terminiert”. Eine Spezifikation $[pre, post]$ aus der Verfeinerungstheorie [34] übersetzt sich in die Prädikatensemantik als: $pre \Rightarrow (t \wedge post)$. Wir können erstere als syntaktischen Zucker ansehen und alle Ausdrücke des Verfeinerungskalküls erhalten so eine Bedeutung, alle seine Gesetze lassen sich aus den Axiomen der prädikativen Semantik herleiten!

Im nächsten Abschnitt sehen wir, dass diese semantische Grundlage auch hervorragend dazu geeignet ist, Terminierungsbeweise mit Programmherleitungen zu kombinieren oder getrennt davon durchzuführen — je nach Bedarf.

4.2 Eine Praktische Theorie der Programmierung

Ich habe die “historischen Theorien” in einer Reihenfolge präsentiert, die einen klaren Fortschritt erkennen lässt: immer einfacher und immer praktischer. Jetzt kommt sozusagen die Krone der Schöpfung, die modernste aller Theorien. Während andere Theorien der Programmierung eher ein wissenschaftliches Spezialgebiet darstellen, ist diese Theorie anwendungsnah: sie gehört (zukünftig) zum Grundlagenwissen eines jeden Informatikers. Weil die Bezeichnung “beste Theorie der Programmierung” vermessen wäre und ihr Autor Bezeichnungen nach Personen nicht mag, nenne ich sie einfach APTOP, so wie sein Buch [16]. Hier ihre Highlights:

1. Dank prädikativer Semantik hat sie eine klare Verbindung zu realen Welt.
2. Dank algebraischer Semantik ist sie vollständig formal, kommt ohne Meta-Logik aus und ist in sich abgeschlossen.
3. Sie enthält eine Programmiersprache, die dank *rekursiver Verfeinerung* noch einfacher ist als strukturierte Programmierung.
4. An Stelle von Terminierungsaussagen beweist man direkt Laufzeitaussagen und das in ebenso formalem Rahmen!

5. Sie umschließt fast alle Bereiche der Programmierung, nicht nur imperativ sequentielle und funktionale, sondern auch parallele, interaktive und sogar probabilistische.

Das Lehrbuch [16] behandelt ganz APTOP auf nur 174 Seiten, da kann sich also jeder selbst weiterbilden. Ich will hier nur auf einige Besonderheiten eingehen.

4.2.1 Algebra

Die Formalisierung der Logik geschieht über Axiome für die Bool'schen Operatoren, wie auch schon in [10]. Besonders einfach ist Hehner's Formalisierung der Mengentheorie (um die sich andere Theorien oft drücken). Schauen wir uns mal die Syntax von Mengen an:

$$\begin{aligned} \text{set} &\rightarrow \text{"}\{ \text{" bunch "}\text{"} \\ \text{bunch} &\rightarrow \text{expression} \\ &\quad | \text{expression "}, \text{" expression} \end{aligned}$$

Damit sind also $\{3, 2, 1\}$ und $\{2, 2, 3, 1\}$ Ausdrücke für Mengen. Wie können wir leicht beweisen, dass die beiden Mengen gleich sind? Antwort: wir betrachten das Komma $,$ als Operator mit den folgenden Axiomen: $(a, b), c = a, (b, c)$ (Assoziativität: wir lassen also im Folgenden immer die Klammern weg), $a, b = b, a$ (Kommutativität: wir können also immer die kleinsten Elemente zuerst schreiben), und schließlich $a, a = a$ (Idempotenz: wir können gleiche Elemente weglassen). Diese Axiome angewendet, sind beide Ausdrücke gleich $\{1, 2, 3\}$ und damit beide gleich.

Wenn man diese Axiome anwendet, stellt man fest, dass man eigentlich nur mit den "Inhalten von Mengen" rechnet und nicht mit den Mengen selbst. Man braucht eigentlich gar keine Mengen mehr, sondern nur noch *bunches*! (Für die deutsche Übersetzung von [16] wird noch nach einem passenden Begriff für dieses neue Konzept gesucht. Vorschläge bitte an mich!) Das Komma ist dann gleichzeitig der Operator für *bunch union*, also die Vereinigung, und für die *bunch intersection* schreiben wir ein Hochkomma. Weil *bunches* "keine Grenzen" haben, sind die Element-von- und Untermengenrelationen ein und derselbe Operator: statt $1 \in \{1, 2, 3\}$ und $\{1\} \subseteq \{1, 2, 3\}$ schreibt man schlicht $1 : 1, 2, 3$. Für Informatiker, die schon ihr Leben lang an Mengen gewöhnt sind, ist das sicher ein Kulturschock. Aber Spezifikationen, Programme und Beweise werden dadurch wirklich einfacher: die Programme nicht weniger realistisch und die Spezifikationen nicht weniger verständlich.

4.2.2 Rekursive Verfeinerung

Ich gebe hier mal ein Beispiel aus [12] wieder. $x' = x * y$ ist eine Spezifikation, die sagt "Der Nachher-Wert von x ist der Vorher-Wert von x mal jenem von y ." Wir wollen also imperative Multiplikation implementieren (und nehmen damit an, der Operator $*$ ist noch nicht implementiert.)

Die Spezifikation sagt nichts über Programmterminierung und ist daher allein recht unnütz. Wir geben zunächst eine Implementierung an und kümmern uns um die Terminierung im nächsten Abschnitt.

Hier ist die Implementierung:

$$\begin{aligned} x' = x * y &\Leftarrow s := 0 ; s' = s + x * y ; x := s \\ s' = s + x * y &\Leftarrow \text{if } y = 0 \text{ then ok} \\ &\quad \text{else if even}(y) \text{ then } (x := x * 2 ; y := y/2 ; s' = s + x * y) \\ &\quad \text{else } (s := s + x ; x := x * 2 ; y := (y - 1)/2 ; s' = s + x * y) \end{aligned}$$

Wir erinnern uns, dass das Implikationszeichen für die Verfeinerung steht, links haben wir Spezifikationen und rechts Programme. Wir stellen fest, dass die Spezifikationen $s' = s + x * y$ auch mehrmals auf der rechten Seite vorkommt. Das ist die rekursive Verfeinerung! Man kann damit Schleifen ohne spezielle neue Regeln ausdrücken. Die gezeigten Implikationen sind normale logische Ausdrücke, die man beweisen kann (machen wir gleich), aber für einen Compiler sind

sie einfach nur Namen. Er sieht das Programm so:

$$\begin{aligned} P &\Leftarrow s := 0 ; Q ; x := s \\ Q &\Leftarrow \text{if } y = 0 \text{ then } ok \\ &\quad \text{else if } even(y) \text{ then } (x := x * 2 ; y := y/2 ; Q) \\ &\quad \text{else } (s := s + x ; x := x * 2 ; y := (y - 1)/2 ; Q) \end{aligned}$$

Für den Beweis des Programms verwenden wir ein einfaches Lemma, dass aus den Definitionen von Zuweisung und Sequenzierung folgt: $(x := e ; P) = (\lambda x \cdot P)e$ (= “ x in P ersetzt durch e ”).

Für die erste Verfeinerung ergibt sich dann:

$$\begin{aligned} &s := 0 ; s' = s + x * y ; x := s \\ = &s := 0 ; x := s + x * y \\ = &x := 0 + x * y \end{aligned}$$

Was wohl sowieso einfach zu sehen war. Für die zweite Verfeinerung gehen wir fallweise vor. Die Semantik des `if` ist ja eine Konjunktion, also hat unsere Behauptung die Form $Q \Leftarrow a \wedge b \wedge c$. Wir können also $Q \Leftarrow a$, $Q \Leftarrow b$ und $Q \Leftarrow c$ einzeln zeigen. Beginnen wir mit dem Fall $y \neq 0 \wedge even(x)$:

$$\begin{aligned} &x := x * 2 ; y := y/2 ; s' = s + x * y \\ = &x := x * 2 ; s' = s + x * (y/2) \\ = &s' = s + x * 2 * (y/2) \\ = &s' = s + x * y \end{aligned}$$

Fast genauso leicht ist der Fall $y \neq 0 \wedge \neg even(x)$ (alle Ersetzungen auf einmal):

$$\begin{aligned} &s := s + x ; x := x * 2 ; y := (y - 1)/2 ; s' = s + x * y \\ = &s' = s + x + x * 2 * (y - 1)/2 \\ = &s' = s + x + x * (y - 1) \\ = &s' = s + x * y \end{aligned}$$

Letzlich fehlt uns nur noch der Basisfall:

$$\begin{aligned} &y = 0 \wedge ok \\ = &y = 0 \wedge y = y' \wedge x = x' \wedge s = s' \\ \Rightarrow &y = 0 \wedge s = s' \\ = &y = 0 \wedge s' = s + x * 0 \\ = &s' = s + x * y \end{aligned}$$

Und damit ist die Verfeinerung bewiesen!

4.2.3 Laufzeitschranken

Die Idee ist wieder genial einfach: wir führen eine Variable t ein, die nicht geändert werden darf und als Prädikat für eine Abstraktion der Zeit steht. “Eine Abstraktion” heißt dabei “eine beliebige Abstraktion ihrer Wahl”. Man kann sich aussuchen, welche Operationen wieviel Zeit kosten sollen, und ändert dann einfach die Axiome dieser Operationen, so dass t als Seiteneffekt inkrementiert wird! Bei Sortieralgorithmen kann man zum Beispiel Vergleiche zählen, bei numerischen Algorithmen zählt man elementare Additionen und Multiplikationen. Im einfachsten Fall rechnet man nur rekursive Verfeinerungen (also Schleifen und Routinenaufrufe) mit jeweils einer Einheit ab. Zum Beweis von Laufzeitschranken braucht man dann Schleifen- bzw. Rekursionsvarianten (also Induktionshypothesen) genauso wie in der axiomatischen Methode. Für den gleichen Aufwand erhält man aber als Ergebnis nicht nur die Terminierungsaussage (die ja so wichtig ist!), sondern auch gleich noch die *asymptotische Laufzeit* des Algorithmus! Nur selten braucht man mehr.

Wir können dies auf das Beispiel anwenden:

$$\begin{aligned} Q &\Leftarrow \text{if } y = 0 \text{ then } ok \\ &\quad \text{else if } even(y) \text{ then } (x := x * 2 ; y := y/2 ; t := t + 1 ; Q) \\ &\quad \text{else } (s := s + x ; x := x * 2 ; y := (y - 1)/2 ; t := t + 1 ; Q) \end{aligned}$$

Wenn wir für Q die folgende Zeitspezifikation einsetzen, erhalten wir wieder eine Bool'sche Aussage, die wir leicht beweisen können. Die Zeitspezifikation ist

$$\begin{aligned} &\text{if } y < 0 \text{ then } t' - t = \infty \\ &\text{else if } t = 0 \text{ then } t' - t = 0 \\ &\text{else } t' - t \leq 1 + \log_2 y \end{aligned}$$

Auch hier beweisen wir wieder fallweise, zur Illustration hier der Fall $y \neq 0 \wedge \text{even}(x)$:

$$\begin{aligned} & x := x * 2 ; y := y/2 ; t := t + 1 ; t' - t \leq 1 + \log_2 y \\ = & t' - (t + 1) \leq 1 + \log_2(y/2) \\ = & t' - t \leq 1 + \log_2 y \end{aligned}$$

Wir haben erst die Ersetzungsregel angewendet und dann einfache Mathematik!

4.2.4 Routinenaufruf

Ich hatte zur axiomatischen Semantik schon erwähnt, dass der Routinenaufruf dort nicht trivial ist. Die Verfeinerungstheorie liefert auch eine Begründung, warum *call-by-reference* und *call-by-name* große Schwierigkeiten machen: wenn man sie verwendet, ist die Verfeinerung nicht mehr kompositional: wenn man den Code einer Prozedur verfeinert, ist das für den Aufrufer “sichtbar”, es ändert sein Verhalten. *Call-by-reference* ist also keine gute Sache, das ist eine theoretische Begründung, warum er in modernen Programmiersprachen nicht mehr vorkommt!

Zugegeben: Man erhält denselben Effekt ja durch Referenz-Typen, aber dann ergibt sich die Semantik der Übergabe aus deren Regeln. Diese behandeln wir übrigens informal im nächsten Kapitel.

Es bleibt uns noch *call-by-value*, also die Wertübergabe. Deren Verhalten kann man leicht mit Unterstützung von Zuweisungen definieren: Gegeben die Deklaration (in EIFFEL-Syntax) `proc(a : A; b : B; ...) is P end`, so hat der Aufruf `proc(x, y, ...)` die Semantik `a := x ; b := y ; ... ; P` (wobei die Variablen a, b, \dots so umbenannt werden, dass sie nicht mit Namen aus dem umgebenden Code kollidieren). An Stelle von Substitutionen (wie bei der mathematischen Funktions-Anwendung) verwendet man also ganz einfach Zuweisungen!

Dies ist nicht nur praktisch sehr nützlich, sondern auch eine theoretische Erklärung für einige Sachen: in EIFFEL werden die Parameterübergabe an Routinen und die Zuweisung gemeinsam als *reattachment* bezeichnet, und es gelten die gleichen Regeln für Typ-Konformität und -Konvertierung. Und in pur-funktionalen Sprachen verwendet man oft zusätzliche Funktionsparameter in endrekursiven Funktionen: man simuliert also damit Schleifen und Zuweisungen.

4.2.5 Schlussfolgerungen

Am Anfang bei den historischen Theorien haben wir gesehen, wie gut und direkt diese sich auf die Praxis ausgewirkt haben: Algol und seine Stapel, Zusicherungen und strukturierte Programmierung, algebraische Spezifikationen von ADTs. Bei den letzteren beginnt aber schon ein Bruch: Theorie und Praxis passen nicht mehr so recht zusammen (weil man imperative Objekte funktional zu spezifizieren versucht). Als in den Anfängen der Informatik noch die elementarsten Grundlagen erforscht wurden, waren Theorie und Praxis kaum zu trennen. Berühmte Wissenschaftler arbeiteten an kommerziellen Programmiersprachen mit, wichtige Artikel erschienen noch im CACM. Die Informatik hat sich aber so schnell entwickelt, dass die Theorie nicht mehr hinter der Praxis her kam, oder anders gesagt: neue Theorien werden von einer Unterwissenschaft entwickelt, aber die andere Unterwissenschaft der Informatik arbeitet noch mit alten Theorien. Immer noch algebraische Spezifikationen und haufenweise Metalogik in der Verifikation von objektorientierten Programmen! Und auch die Vorlesungsinhalte an Universitäten wurden zu einer Zeit festgelegt, als die besten Theorien noch nicht entdeckt waren.

Die Sprache EIFFEL verkörpert den letzten Stand der Theorie in der Praxis: Abstrakte Datentypen und axiomatische Semantik. Wir befinden uns in den 80er Jahren! Es ist eines der “Löcher” von denen ich am Anfang der Arbeit sprach, dass aktuelle Programmiersprachen noch nicht die Verfeinerung der 90er Jahre beherrschen. (Ausnahme ist B, aber nur halb, denn ihre Verfeinerung ist auch noch sehr “80er”.) Ganz so als ob das nicht schon schlimm genug wäre, verwenden aber viele aktuelle Forschungsprojekte (also Wissenschaftler, die es eigentlich wissen sollten!) noch die Technologie der 70er Jahre (in Form von JAVA, einer Sprache ohne Verträge). Es ist also eine wichtige Aufgabe, APTOP unter den Informatikern weiter zu verbreiten. Als Belohnung winkt uns die Erfüllung von Dijkstras Prophezeiung, also eine schöne neue Welt!

5 Ansätze einer allgemeinen Theorie

5.1 Versprechen und Grenzen des Formalen Ansatzes

Wenn man manche Bücher über den formalen Ansatz des Programmierens liest, kann man leicht ins Schwärmen geraten: Das Programmieren wird dabei auf einen reinen Rechenprozess zurückgeführt, den man leicht erlernen kann und dessen Ergebnis leicht überprüfbar ist, weil es in ganz einfachen Schritten erreicht wurde. Im Idealfall läuft es wie bei den “Textaufgaben” der Schulmathematik: “Übersetzen Sie die gegebenen Informationen in Gleichungen; lösen Sie die Gleichungen; antworten Sie in einem Satz!” [15]

Formalisten stellen den Weg von der Spezifikation zum Programm als eine Transformation dar, bei der nur logische und mathematische Gesetze angewendet werden. Diese Sichtweise unterschlägt aber einen wichtigen Aspekt. Denn die Gesetze der Logik sind nun mal nur so einfach wie “Wenn $a \wedge b$ gilt, dann gilt auch a und es gilt b .” Damit allein kommt man natürlich nicht weit. Man braucht zusätzliche Informationen, die in der Sprache der Logik ausgedrückt sind. Wenn man in einem Programm mit Zahlen arbeitet, braucht man die Gesetze der Arithmetik und Algebra, für Mengen braucht man Mengentheorie und für Graphen braucht man Graphentheorie. (Natürlich braucht man auch immer die Gesetze der Programmiersprache selbst (die Programmtheorie) aber wie wir im Abschnitt über die Formalen Theorien gesehen haben, sind deren Gesetze vergleichsweise einfach. Man merkt schon bei der Herleitung und Verifikation mittlerer Algorithmen, dass die anderen Theorien den wesentlichen Anteil haben.) Wenn man all das als formal-mathematisches Grundlagenwissen ansieht, dann handelt es sich von der Spezifikation zum Programm wirklich nur um “eine einfache logische Transformation”. Will man aber ein besseres Bild von der formalen Programmentwicklung bekommen, so sollte man alle Theorien außer der Logik (Boolsche Theorie) und der Programmtheorie als gesondertes Anwendungswissen betrachten. Man erhält dann folgendes, genaueres Bild:

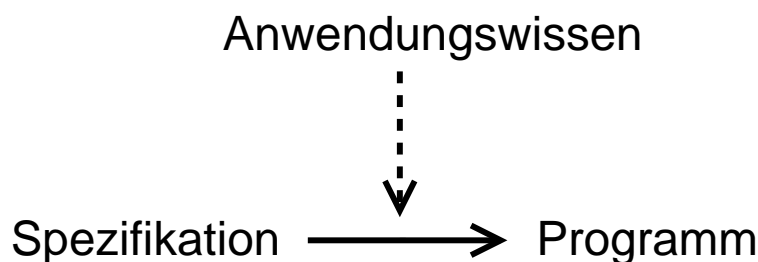


Abbildung 5.1: Softwareentwicklung erfordert Anwendungswissen

Bei den algorithmischen Problemen, die von den Formalisten behandelt werden, stammt das “Anwendungswissen” aus Mathematik und Informatik selbst — es ist also leicht zu formalisieren, ist ja schon fast formal. (In APTOP sind das *bunch*-Theorie und *string*-Theorie.) Im allgemeinen Fall handelt das Anwendungswissen aber von der realen Welt oder von anderen technischen Komponenten, und das lohnt sich meist nicht zu formalisieren. In der Praxis findet man Anwendungswissen höchstens manchmal in operationalisierter Form vor (als fertige Komponente oder als Checklisten für die Entwicklung), aber nicht als formale Theorie. Wissen über technische Komponenten könnte man aber durchaus formal bereitstellen; genau das tut man ja, wenn Komponenten formal spezifiziert sind — hier liegt noch ein großes Entwicklungspotenzial in der Zukunft [33, 41].

Aufgrund des Mangels an formalem Wissen (in Form von Theorien) sind also die reinen formalen Methoden oft nicht anwendbar. An anderen Stellen wiederum schießen sie über das Ziel hinaus: Viele der entwickelten Programmierungstechniken wurden mit ihrer Formalisierung auch operationalisiert und können bereits vollautomatisch durchgeführt werden. So können zum Beispiel einige Compiler bereits asymptotische Verbesserungen in Programmeffizienz erzielen: also nicht nur um einen konstanten Faktor schneller, sondern qualitativ, zum Beispiel von $O(n^2)$ nach $O(n)$. Für Aufgaben, die mein Compiler (oder ein anderes Werkzeug) für mich übernimmt, brauche ich natürlich keine Methode mehr. Dies ist die zweite Grenze des rein formalen Ansatzes.

5.1.1 Voll-Formal versus Halb-Formal

Eine voll-formale Vorgehensweise würde erfordern, dass man alle zulässigen elementaren Schritte vorher formal festlegt, und dann die gesamte Herleitung auf diese elementaren Schritte zurückführt. (Man kann natürlich zusammengesetzte Schritte mit Hilfe der elementaren definieren, wenn einer der elementaren Schritte “Anwendung eines zusammengesetzten Schritts” ist.) Dies hat den Vorteil, dass man absolut kein eigenes Einschätzungsvermögen mehr braucht (“Ist das offensichtlich oder nicht?”) und dass alle Annahmen eines Beweises explizit gemacht werden.

Die halb-formale Sicht setzt dagegen, dass man nur soweit formalisieren sollte, bis man sich selbst und andere überzeugt hat, dass das eigene Vorgehen fehlerfrei ist. (Natürlich meinen die Vertreter dieses Ansatzes, für diese Überzeugung mehr Formalismus zu brauchen als andere!) In diesem Zusammenhang hat Harlan Mills [36] geschrieben, “ein Beweis ist ein kontrolliertes Experiment der Überzeugung einer Person”. Zum Nutzen von formalen Argumenten in diesem Überzeugungsakt führt man wieder an, dass diese erstens objektiver sind und zweitens oft kürzer.

Die halb-formale Sicht bezeichnet man auch als “rigoros” (engl. *rigorous*, vom lateinischen *rigor* = streng). Ihr Vorteil ist, dass man damit nicht nur Überzeugung transportiert, sondern auch ein Verständnis für den Algorithmus. “Warum funktioniert es?” und “Wie funktioniert es?” sind ähnliche Fragen, und diese Informationen sind nützlich, um später Anpassungen am Programm vorzunehmen. Der halb-formale Ansatz überlässt es dem Autor, wie klein er seine Schritte wählt, und was er alles als “trivial” bzw. “offensichtlich” ansieht. Dabei ist es bequem, dass man beliebig komplexe Schritte weglassen kann, beziehungsweise dass man in jedem Schritt beliebig viel Anwendungswissen verwenden kann. Dies erfordert natürlich, dass sowohl Autor als auch ein kritischer Leser, Experten in der Domäne sind – ansonsten macht man womöglich implizit Annahmen, die dann in der Dokumentation fehlen (Überraschung in der Wartung) oder die gar nicht stimmen (Überraschung im Test oder in der Produktion)!

Solche Überraschungen kann man mit einem voll-formalen Ansatz vermeiden: dann muss man all die lächerlichen Details explizit angeben und jede Annahme wird offengelegt. Allerdings ist das auch ein riesiger Aufwand, und Irrtümer können ja trotzdem noch passieren. Eine wirklich voll-formale Vorgehensweise lohnt sich also nur, wenn sie auch mechanisiert ist: der Computer muss bei der Erstellung der kompletten Beweise behilflich sein, und ihre absolute Korrektheit garantieren.

Damit sich voll-formale Entwicklung in der Praxis lohnt, braucht man mindestens:

1. Ein formales System, dessen elementare Observationen perfekt auf die Phänomene des Anwendungsbereichs passen.
2. Software-Tools, die genau so viele elementare Schritte automatisch durchführen können, wie ein Ingenieur als “trivial” bezeichnen würde.

Punkt eins ist ganz wichtig, denn wenn man mit Computer-überprüften Beweisen arbeitet, ist die Fehlerrate so gering, dass plötzlich die Begriffsbestimmungen und der (sonst eigentlich triviale) informale Anteil das schwächste Glied in der Kette bilden. Dieser Punkt ist auch Voraussetzung für eine erfolgreiche halb-formale Vorgehensweise und wir haben bereits gesehen, dass dies dem formalen Vorgehen generell eine untere Schranke setzt.

Punkt zwei heißt nichts anderes, als dass der Computer beim kritischen Lesen nicht dümmer sein sollte als ein Experte, der sich dumm stellt. Die einzige mir bekannte Anwendung von mechanisierten Beweisen in Industrie-Projekten (außer auf Schaltkreis-Niveau, wo eine Vielzahl von Methoden verfügbar ist) fand mit der B-Methode statt, und aus diesem Projekt wird berichtet, dass der Aufwand für die Computer-gestützte Erstellung von Beweisen immer noch um ein Vielfaches höher war als für die Überzeugung von Menschen. Ein solcher Aufwand lohnen sich bisher nur bei hochkritischen Systemen. Andererseits hat B auch gezeigt, was mit dem Stand der Technik möglich ist. Mittlerweile gibt es ja schon bessere Techniken als B (welches weder prädikative noch algebraische Semantik verwendet), deswegen kann man hoffen, dass man mit dem Computer bald sogar besser beweisen kann als ohne, weil er sogar einige “einfache” (nicht mehr nur “triviale”) Dinge selbst herausfinden kann. In den natürlichen Grenzen, die dem formalen Ansatz durch Punkt eins gesetzt sind, steht uns vielleicht bald gute Software-Unterstützung zur Verfügung. Da das aber hier und jetzt noch nicht der Fall ist, schaut sich diese Arbeit noch nach anderer Art von Maschinenunterstützung um.

Computer-geprüfte Beweise und Computer-erstellte (Teil-)Beweise fügen den formal-logischen Aspekten (die für ersteres nötig sind) noch einen algorithmischen Aspekt hinzu. Darauf muss man beim Entwurf (oder der Auswahl) eines formalen Systems Rücksicht nehmen; möglicherweise muss man die Programmiersprache und ihre Semantik ganz neu entwerfen, um die Mechanisierung effizient möglich zu machen. Man sieht das ja teilweise schon an AP-TOP, die sich durch ihre *bunches* und rekursive Verfeinerung bereits sehr von herkömmlichen Programmier- und Spezifikations-sprachen unterscheidet.

5.1.2 Methode informal?

Wenn man den Blick vom formalen Ansatz auf populäre “Methoden” der Softwareentwicklung richtet, stellt man fest, dass sich diese weit geringeren Ansprüche stellen. Die populärsten Ausdrucksmittel sind Diagramme, deren Inhalte zum großen Teil rein strukturell sind, also keine Funktionalitäten (in der Spezifikation) oder Schnittstellen (im Entwurf) enthalten. Solche Methoden mögen den Entwicklern nützliche Hinweise geben zur Strukturierung des Entwicklungsprozesses und zur Erstellung der Lösung — aber sie geben eben keine Antworten auf die essenziellen Fragen: wie erfülle ich die (funktionalen) Anforderungen des Programms? Woher weiß ich, dass das Programm auch tut, was es soll? Und dass mein Entwurf zu einem solchen Programm führt?

Durch die Entwurfsmuster hat die OO-Literatur zum ersten Mal klare Aussagen über gute und schlechte Programme gemacht, aber diese sind immer noch zu beispielhaft, um von einer Methode zu sprechen. Immer noch zu sehr wie “Hey, das hat bei uns gut funktioniert, vielleicht passt es in Dein Programm auch.”

Gerade weil die Standard-OO-Literatur (außer Meyer) Verträge nur am Rand oder gar nicht erwähnt, kann sie zwar viel über ADTs und Abstraktion reden, deren Essenz aber nicht erfassen. Die versteckten Informationen des *information hiding* sind ja gerade die Differenz aus Spezifikation und Implementierung! Ohne explizite Spezifikationen gibt es also auch keine wirkliche Abstraktion. (Diese wird im nächsten Kapitel behandelt.)

5.1.3 Herausforderungen einer modernen Programmiermethode

Als sich in den Anfängen der Informatik herausstellte, dass die Programmierung von Computern wesentlich mehr ist, als das Übersetzen von Anwendungsregeln in Code, hat sich das Gebiet der Programmiermethodik gebildet und man begann sich zu überlegen, wie man denn nun am besten programmieren sollte. Dabei wurden zunächst Algorithmen und Datenstrukturen untersucht (die wenig später ein eigenes Fachgebiet bildeten), viel wurde geschrieben über Programmstrukturierung und -entwurf: Programme sollten zunächst in einer Pseudo-Programmiersprache (PDL) beschrieben werden, und dann in die Programmiersprache übersetzt, Programme sollten in Routinen und Module aufgeteilt werden, Variablen sollten mit dem Typ der enthaltenen Daten

gekennzeichnet werden, “verzichten Sie auf selbstmodifizierenden Code, benutzen Sie GOTOs nur für dokumentierte Schleifen und bedingte Anweisungen...”

Wenn man sich diese Liste anschaut, fällt schnell auf: vieles was früher zum “guten Programmierstil” gehörte, ist heutzutage bereits in Programmiersprachen realisiert! Man muss heutzutage niemandem mehr sagen, er soll GOTOs und selbstmodifizierenden Code vermeiden, weil es gar nicht mehr direkt möglich ist, diese überhaupt zu verwenden. Module und Typen muss man nicht mehr bewerben, weil sie sich dem Programmierer ganz offensichtlich anbieten. (Man könnte zwar allen Code in eine Klasse schreiben und alle Variablen vom Typ ANY deklarieren (der Spitze der Subtyping-Hierarchie), um diese neuen Funktionen der Programmiersprache zu umgehen, aber das macht ganz direkt mehr Arbeit, als ein zumindest halbwegs “ordentlicher” Ansatz.)

- Programmiersprachen werden immer mächtiger und übertreffen teilweise schon die Spezifikationsprachen. (Zum Beispiel bei dynamischer Bindung und Modulen.)
- Alle grundlegenden Algorithmen sind bereits in Bibliotheken vorhanden. Und auch für die meisten technischen Angelegenheiten — vom GUI bis zum Netzwerk— gibt es Bibliotheken.
- Werkzeuge übernehmen viele der Aufgaben, die früher gute Programmierer auszeichneten: Generierung von Schnittstellen-Beschreibungen und Diagrammen, automatisches Beachten der Übersetzungsreihenfolge.

Während Programmierer all diese Vorteile genießen, die ohne weiteren Aufwand zur Qualität und Produktivität beitragen, wird aber oft ignoriert, dass man noch viel mehr erreichen kann, wenn man ein bisschen Aufwand betreibt. Beim Lesen von Programmen hatte ich oft den Eindruck, dass Programmierer die modernen Vorteile nur nutzen, wenn es “wirklich ganz offensichtlich” ist. Typen und Module werden aber oft rein intuitiv und unreflektiert verwendet, Vererbung findet nach Gefühl statt. Auch Kommentare und Zusicherungen findet man nur sporadisch.

Andererseits stellen moderne Programmiersprachen auch ganz neue Herausforderungen, die es vorher noch nicht gab. Besonders schwerwiegend scheint mir zu sein, dass objektorientierte Programmiersprachen einen so riesigen Gestaltungsspielraum bieten. Natürlich gibt es auch in anderen Programmiersprachen (und Paradigmen) unendliche viele Möglichkeiten jedes einzelne Problem zu lösen. Aber es scheint mir, dass OO-Sprachen durch ihre vielfältigen Strukturierungsmöglichkeiten die Zahl der möglichen Lösungen noch einmal potenzieren. Das erkennt man auch daran, dass die typischen OO-Konstrukte von einem Compiler einfach “wegkompiliert” werden: Modulgrenzen werden ohne Effizienzverlust überschritten. Viele verschiedene OO-Programme beschreiben eigentlich denselben Maschinencode. Auch ist dieses Phänomen in funktionalen oder prozeduralen Programmiersprachen nicht so stark ausgeprägt, weil dort einfache Transformationsregeln existieren, die die Abstände zwischen Elementen des Lösungsraums verringern. Zwei Programme, die das Gleiche tun, sehen nicht so radikal unterschiedlich aus.

Lösung zu diesem Problem ist einerseits das Refaktorisieren, das uns lehrt, welche noch so unterschiedlichen Programme eigentlich ganz ähnlich sind — siehe dazu Abschnitt 7.3, wo wir auch sehen, dass die funktionalen und prozeduralen Refaktorisierungen eigentlich trivial sind (entsprechende Werkzeuge sind gerade dabei, sich auf dem Markt durchzusetzen), die objektorientierten aber noch ziemlich vage. Andererseits hilft uns die bereits in der Einleitung (Abschnitt 1.3) erwähnte Strategie, den Lösungsraum möglichst stark zu beschränken. Es ist wohl so, dass die objektorientierte Programmierung besser geeignet ist, Probleme zu lösen, aber sie erfordert vom Entwickler auch viel mehr Wissen. Und die vorhandene Literatur bezieht sich dann doch zu oft auf die Details einzelner Programmiersprachen oder Diagramm-Sprachen und zu wenig auf die Grundherausforderungen der Objekt-Orientierung.

5.2 Der Dreibund

In der Einleitung haben wir bereits den Begriff der Konsistenz kennen gelernt, der die strenge Vorstellung der Korrektheit ersetzen soll. Es geht also darum, geschickt Redundanz in die Software einzufügen und zwar so, dass sie dem Entwickler möglichst viel nützt und so dass die Konsistenz möglichst leicht automatisch sicher gestellt werden kann. Alle dabei erstellten Dokumente (insbesondere Spezifikationen, Programmcode und Testfälle) werden als ein einziges Ergebnis betrachtet. Eine Sicht auf dieses Ergebnis ist das ausführbare Programm, andere Sichten dienen der Überprüfung, Weiterentwicklung und so weiter. Die drei Grundpfeiler unseres Ansatzes sind:

Spezifikation Die Funktion eines jeden Programmteils wird vollständig, präzise und so einfach wie möglich beschrieben. Entweder in präziser Allgemeinsprache, mit Bool'schen Ausdrücken oder direkt mit Anweisungen.

Entwurf Jedes implementierte Programmstück ist das einfachst-mögliche, dass die Spezifikation erfüllt. Untereinheiten haben selbst wieder sehr einfache Spezifikationen.

Test Zu jedem Programmstück gibt es eine Test-Routine, die genau dann ohne Laufzeitfehler durchläuft, wenn das Programmstück korrekt ist. Gut platzierte Zusicherungen verursachen Laufzeitfehler sehr nahe bei den wirklich fehlerhaften Stellen.

Diese drei Grundbegriffe beschreiben keine Phasen, die man nacheinander durchläuft, sondern Prinzipien, die man immer beachtet. Testfälle sind zum Beispiel nur eine spezielle Art von Spezifikationen (siehe übernächster Abschnitt). Und das bewusste *Entwerfen* bezieht sich auf jede Art von Dokument: Spezifikationen (inklusive Testfälle), Unter-Spezifikationen und Programme. Aber den Begriff der Spezifikation haben wir jetzt schon sehr lange verwendet, ohne uns darüber Gedanken zu machen...

5.2.1 Was sind Spezifikationen?

Es ist leider immer noch die am weitesten verbreitete Definition von "Spezifikation", dass diese das *Was* (tut es?) und nicht das *Wie* (tut es das?) eines Programms beschreibt. Wenn in der Literatur genauere Definitionen angegeben werden, dann sind diese schon wieder speziell auf einen Formalismus zusammen geschnitten: sie sprechen von Zuständen, Ereignissen, etc.) Diese Charakterisierung ist leider sehr unzureichend. Ich kann zum Beispiel sagen: "Was tut das Programm? Es überweist Geld von einem Konto auf ein anderes. Wie tut es das? Es bucht das Geld von einem Konto ab und bucht es auf das andere Konto auf." Aber genauso gut kann ich sagen: "Was tut das Programm? Es bucht Geld von einem Konto ab und bucht es auf ein anderes Konto auf." Zwischen "wie" und "was" gibt es keine gute Grenze!

Das Beispiel enthält außerdem noch den Fehler, dass die zweite Aussage im Wesentlichen nur den Begriff "überweisen" definiert. Und sie verwendet nicht einmal die angebrachte Formulierung "denselben Betrag".

Dieser Wie-Was-Mischmasch ist umso trauriger, weil es wirklich eine sinnvolle allgemeine und trotzdem präzise Definition von "Spezifikation" gibt, nämlich die folgende:

Spezifikation eines Softwareelements Die Spezifikation eines Softwareelements beschreibt das von außen sichtbare Verhalten des Systems. (Wie jede Beschreibung, die Ingenieure anfertigen, tut sie das möglichst einfach und präzise.) Erste Aufgabe der Spezifikation ist es, die Grenze zwischen dem Softwareelement und seiner Umgebung festzulegen, das heißt zu definieren, was im vorliegenden Fall "außen" bedeutet.

Auf den Abschnitt über formale Beschreibungen bezogen, definieren wir die Außengrenze eines Softwareelements über die elementaren Beobachtungen, die für das Element sichtbar sind und von ihm kontrolliert werden. Diese Liste der Beobachtungen ist der notwendige informale Teil, der einer jeden Spezifikation beiliegen muss. Die Theorie der Verfeinerung beseitigt die Wie-Was-Unklarheiten, indem sie klar macht, dass sowohl Spezifikationen als auch Programme eigentlich dasselbe beschreiben: nämlich das Verhalten eines Computers. Der Unterschied ist allein, dass die Spezifikation dies auf beliebige Art tun kann, während Programme sich auf die Programmiersprache beschränken müssen.

Die andere Seite der Außengrenze eines Softwareelements kann sein:

- menschliche Benutzer,
- technische Geräte oder
- andere Software.

Je nach der Art der Umgebung und Art der Grenze sind die auftretenden Probleme und die notwendigen Methoden (und Notationen) völlig unterschiedlich! Bei technischen Geräten stellt sich zum Beispiel die Frage: “Läuft es in Echtzeit (zum Beispiel Lautsprecher)? Oder nicht (zum Beispiel Drucker)?” Bei Schnittstellen zu Menschen ist eine der Fragen: “Ist die Schnittstelle graphisch (Bildschirm)? Oder akustisch (Telefon)?” Je nach Antwort braucht man eine andere Spezifikationsmethode und -Notation. Michael Jacksons Buch [22] zeigt, wie man genau diese Fragen beantwortet, um die richtige Methode für ein Problem auszuwählen. Oder genauer gesagt: die richtige Methode für jedes Teilproblem eines Problems.

Diese Arbeit wird sich mit *inneren Grenzen* (Schnittstellen) von Software beschäftigen. Diese sind im Wesentlichen gleich den externen Schnittstellen zu anderer Software und werden im nächsten Kapitel ausführlich beschrieben.

5.2.2 Spezifikationen versus Tests

Ein einzelner Testfall beschreibt Computerverhalten punktuell: bei dieser Eingabe wird jene Ausgabe erwartet. Damit ist auch ein Testfall eine Spezifikation und wir können mehrere Testfälle mit logischer Konjunktion verknüpfen. Ein so entstehender Satz von Testfällen ist wieder eine Spezifikation. Wenn man die Testfälle direkt auf die elementaren Ein- und Ausgabe-Beobachtungen eines Programms bezieht (und nur dann sind sie ja objektiv eindeutig), dann lassen sie sich auch leicht formalisieren — in der Regel als Einträge in eine Tabelle. Testfälle als Spezifikationen haben zwei wichtige Vorteile: erstens sind sie sehr konkret (eben beispielhaft) und dadurch leicht zu verstehen. Zweitens kann man vollautomatisch “beweisen”, dass ein Programm eine Spezifikation in Form von Testfällen erfüllt: man führt das Programm einfach mit den Eingaben aus und prüft die Ausgaben. Die Nachteile sind, dass ein Satz von Testfällen keine vollständige Spezifikation darstellt, dass er oft länger ist, als eine direkte Spezifikation und letztlich, dass er weniger Hinweise zur Erstellung eines Programms gibt, als eine direkte Spezifikation.

Wenn wir eine Funktion zum Runden von Zahlen spezifizieren, so reichen die folgenden Testfälle aus, um einem Leser mitzuteilen, was wir mit “Runden” meinen:

Eingabe	Ausgabe
0	0
1	1
0.1	0
0.5	1
2.2	2
-0.1	0
-0.5	0
-2.8	-3

Eine direkte Spezifikation für dieselbe Art von “Runden” wäre: “Diejenige ganze Zahl, die höchstens um $\frac{1}{2}$ größer ist oder weniger als $\frac{1}{2}$ kleiner.” Das “oder” in diesem Satz deutet schon auf die Fallunterscheidung zwischen Aufrunden und Abrunden hin, die eine Implementierung machen muss. Übrigens unterscheidet sich eine andere Interpretation von “Runden” im Testsatz nur durch den Fall -0.5 , der dann zum Ergebnis -1 führt. Interessanterweise würde diese kleine Änderung (die für den Leser sofort verständlich ist) in der direkten Spezifikation eine weitere Fallunterscheidung erfordern.

Da Tests inheränt formal unvollständig sind, stellt sich die Frage, wie viele Testfälle man denn braucht, um das Computer-Verhalten mit großer Wahrscheinlichkeit vollständig zu beschreiben. Mit Hilfe von Überdeckungsmaßen kann man sagen, wann Testfälle das Verhalten eines gegebenen Programmtextes (mit großer Wahrscheinlichkeit) komplett beschreiben. Üblicherweise misst man diese Überdeckung beim Testen (ein Werkzeug macht das automatisch), so dass man ein Kriterium hat, wann ein Testsatz ausreichen umfangreich ist. Auf der anderen Seite ist mir aber keine Untersuchung bekannt, die fragt wie viele Testsätze man braucht, damit ein menschlicher Leser die Programmfunktion eindeutig aus den Testfällen herauslesen kann. Dabei wäre dies gerade für das Extreme Programmieren wichtig, wo ja die Testfälle einzige Dokumentation (außer dem Quellcode) sind. Da unsere Methode aber sowieso auch immer direkte Spezifikationen einsetzt, kommen wir auch ohne diese Untersuchung aus.

Übrigens: wenn ein Satz von Testfällen wider Erwarten nicht die gewünschte angezeigte Überdeckung erreicht, so kann das auch daran liegen, dass der Programmcode viel komplizierter ist als eigentlich notwendig oder dass ein größeres Missverständnis vorliegt. In jedem Fall kann eine direkte Spezifikation helfen, das Problem zu klären.

5.2.3 Testen von Spezifikationen

Extreme Programmierer bevorzugen Testfälle gegenüber direkten Spezifikationen, weil erstere eine kanonische automatische Konsistenz-Prüfung haben: man muss nur einen Testtreiber schreiben, und dann bekommt man schnell ein eindeutiges Ergebnis — so oft man möchte. DESIGN BY CONTRACT mit seinen Zusicherungen im Programmcode erreicht aber dasselbe und stellt sogar die Sprache zur Verfügung, in denen man Spezifikationen ausdrücken kann. Für Testfälle muss man dagegen stets selbst ein Format entwerfen und den Testtreiber dazu schreiben. (Oft ist der Testtreiber nur eine Klasse, und die Testfälle schreibt man als Aufrufe. Dies ist in vielen Fällen ausreichend, oft aber auch zu sporadisch, da es zu viel Redundanz enthält.)

Professionellerweise unterscheidet man das Testen in die Auswahl der Eingabedaten und in die Überprüfung der Ergebnisse. Bisher haben wir einfache Testfälle betrachtet, bei denen Eingaben einzeln gegeben waren, so dass man die Ausgaben ebenso direkt daneben schreiben konnte. Wenn man aber Testdaten automatisch massenweise generieren will, braucht man eine andere Möglichkeit, die passenden Ergebnisse zu spezifizieren. Eine Möglichkeit ist, das Programm zusammen mit einem zweiten laufen zu lassen (das dieselbe Spezifikation implementiert) und die Ergebnisse zu vergleichen. Man muss aber nicht unbedingt die gewünschten Ergebnisse generieren und dann vergleichen. Man kann ja auch einfach die erhaltenen Ergebnisse überprüfen! Und genau das machen Spezifikationen (genauer gesagt: Nachbedingungen) im DESIGN BY CONTRACT.

Dieses professionelle Testprinzip wurde in der QUICKCHECK-Bibliothek [28] für Haskell vorbildlich umgesetzt. Es gibt dort Generatoren, die Eingabewerte verschiedener Datentypen automatisch (pseudo)zufällig generieren, und es gibt eine in Haskell eingebettete Sprache zur Beschreibung von algebraischen Eigenschaften — ganz ähnlich den Nachbedingungen von DESIGN BY CONTRACT. Die algebraischen Eigenschaften sind dabei gleichzeitig direkte Spezifikationen (möglicherweise selbst mit einer gewissen Redundanz, die beim Testen auf Konsistenz geprüft wird) und sie sind Teil der Tests. Diese Art von Tests durch *Generierung* (von Eingaben) und *Überprüfung* (von Ausgaben) ist nicht nur eine sinnvolle Alternative zu fallweisen Tests, sondern auch notwendig um Zusicherungen im Programmcode (insbesondere Vorbedingungen) auf

Konsistenz zu überprüfen. Bei manchen komplizierten technischen Implementierungen lassen sich viele Fehler ausschließen, wenn nur die Einhaltung der Invariante geprüft wird. Wenn diese Invariante als Zusicherung formuliert wurde, reicht schon eine einfache Generierung von Eingabewerten, um automatisch auf all diese Fehler zu prüfen. Wir sehen hier, wie sich Tests und Spezifikationen ergänzen. Tests müssen ein Teil der Methode sein, denn erst mit hinreichend vielen Eingaben entfalten auch die Laufzeitprüfungen der Konsistenz ihre volle Wirkung.

5.2.4 Ziele von und Vorgehen bei Tests

Wie man Tests am besten durchführt wurde, bereits vielfältig in der Literatur beschrieben. Diese Arbeit geht nur näher auf die Rolle von Tests in der Methode des Programmierens ein. Daraus ergibt sich, dass wir nur Tests einzelner Programnteile behandeln, denn große System-Tests oder gar Performance-Tests gehören nicht mehr zu den Aufgaben eines einzelnen Programmierers.

Als Ziel von Tests wird üblicherweise das “Finden von Fehlern” angegeben. Ich stimme aber mit Peter Schaumann [26] darin überein, dass diese Sichtweise zwar teilweise nützlich, aber gesamt-methodisch auch sehr irreführend ist. Natürlich ist das Finden von Fehlern die große persönliche Motivation von Testern, man muss darauf aus sein, Fehler zu finden, um erfolgreich zu testen. Und man sollte darauf vorbereitet sein, auch Fehler zu finden, und um sie zu beseitigen. Wenn man dies aber als alleiniges Ziel betrachtet, wird möglicherweise sehr viel Programmierarbeit in die Test-Phase verlagert — und dass unterläuft natürlich jeden Projektplan und kann auch den Gesamtaufwand erheblich erhöhen. Es gehört ja gerade zur Schlumper-Methode, viel Programmierarbeit in die Testphase zu verlagern, wo doch eigentlich schon alles fertig sein sollte. Wenn man viele Fehler findet, hat man definitiv vorher zu schlecht gearbeitet!

Wie Schaumann sehr schön beschreibt, ist das eigentliche Ziel das Messen der Restfehler-Anzahl. Oder anders gesagt: wir wollen mit hoher Wahrscheinlichkeit sicherstellen, dass die Software keine gravierenden Fehler mehr enthält. Und solch eine “Versicherung” braucht der professionelle Softwareanbieter. (Leider haben die wenigsten Software-Firmen einen Ruf zu verlieren, wenn sie fehlerhafte Produkte ausliefern! Aber es gibt tatsächlich Firmen (darunter sd&m, die das zitierte Buch herausgeben), die ihre Software mit Wartungsgarantien liefern. Das heißt der Lieferant trägt die (sehr hohen) Kosten zum Beseitigen von Fehlern, die erst nach Inbetriebnahme gefunden wurden. Und das ist eine starke Motivation zum effektiven Testen.)

Über die Reihenfolge der Dokument-Erstellung ist man sich aber einig. Zitat aus [26, Seite 291]:

“ Wichtig ist hier das Denken von außen nach innen: Zuerst werden zur spezifizierten Schnittstelle die Testfälle erstellt, dann der Testtreiber geschrieben und erst zuletzt, [...] der eigentliche Code programmiert. ”

Die Spezifikation dient hier also als erste Instanz und auch als Bindeglied zwischen Tests und Programmcode. Sie legt nicht nur auf knappe und präzise Art die gewünschte Semantik fest, sondern auch die Form einer Schnittstelle: Prozedur oder Funktion, formaler Typ der Parameter, ... Mehr dazu im nächsten Kapitel.

5.3 Die Operationalisierungshierarchie

Hier ist das Dogma der Verfeinerungsbewegung (aus [12], aber man findet es ähnlich in [34]):

“ A specification serves as a contract between a client who wants a computer to behave a certain way and a programmer who customizes a computer to behave as desired. For this purpose, a specification must be written as clearly, as understandably, as possible. The programmer then refines the specification to obtain a program, which a computer can execute. Sometimes the clearest, most understandable specification is already a program. When that is so, there is no need for any other specification, and no need for refinement. However, the programming notations are only part of the

specification notations: those that happen to be implemented. Specifiers should use whatever notations help to make their specifications clear, including but not limited to programming notations. ”

Das Ziel der Programmierung nach dieser Theorie ist die Beseitigung aller Spezifikationsnotationen aus dem Programm. Dadurch wird die Spezifikation ausführbar gemacht, in einem Wort: *operationalisiert*. Wenn man annimmt, dass eine formale Spezifikation eine notwendige Etappe auf dem Weg vom menschlichen Verlangen zum ausführbaren Programm ist, dann ist logischerweise der Schritt von der formalen Spezifikation zum Programm ein ganz wesentlicher Gegenstand, und die Forschung ist völlig praxisgerecht, wenn sie nur diesen Schritt untersucht. Die klassischen Formalisten legen also folgenden (imaginären) Prozess zugrunde:

Menschliches Verlangen \rightarrow Formale Spezifikation \rightarrow Ausführbares Programm.

Die Verfeinerungstheorie ergänzt dies, indem sie es erlaubt, den Zwischenschritt zu überspringen, wenn die Formalisierung der Anforderungen bereits ausführbar ist.

Menschliches Verlangen \rightarrow Formale Spezifikation
 \rightarrow Ausführbares Programm

Wir gehen also davon aus, dass die Prozesse des *Operationalisierens* und des *Formalisierens* vermischt sind. Außerdem sind für uns nicht alle formalen Spezifikationen gleich, weil sich nur manche in der automatisch prüfbareren Zusicherungssprache ausdrücken lassen. Dadurch kommen wir zum neuen Bild von Abbildung 5.2.

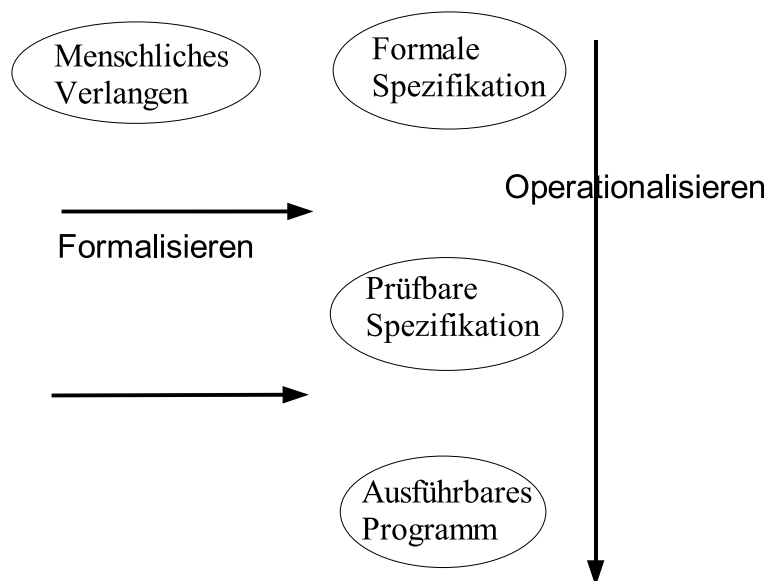


Abbildung 5.2: Vom menschlichen Verlangen zu ausführbarem Code

Überprüfbare Spezifikationen stehen also zwischen den ganz allgemeinen formalen Beschreibungen (oben) und den ausführbaren Beschreibungen (Programme, ganz operational, unten). In manchen Fällen muss man wirklich schon einen Verfeinerungsschritt machen, um von der einfachst-möglichen Spezifikation zu einer zu kommen, die gleichzeitig auch überprüfbar ist.

Die genauen Grenzen zu überprüfbaren und ausführbaren Spezifikationen hängen von der gewählten Programmiersprache ab. Die folgende Definition des Prädikats *prim* kann als nicht-überprüfbar angesehen werden: $prim(x) \equiv \neg \exists p \in \mathbf{N} - \{1, x\} \cdot divides(p, x)$. Wenn wir den Bereich für die Variable p einschränken, erhalten wir eine prüfbare Definition: $prim(x) \equiv \neg \exists p \in \mathbf{N} \cdot 1 < p < x \wedge divides(p, x)$. Man kann diese Definition auch schon benutzen, um Primzahlen zu generieren, aber dazu sollte man sie noch etwas weiter verfeinern — mindestens bis zum Sieb von Eurasthenes.

Außerdem erlaubt es unsere Operationalisierungshierarchie formal/informal gemischte Spezifikationen zu haben. Die formalen Spezifikationen sind dann *unvollständig* und damit als abschließliches Testmaterial nicht ausreichend. Trotzdem sind sie natürlich noch eine gute Dokumentation und im Einsatz als Zusicherungen können sie noch eine Menge Fehler finden. Formal gesehen, besteht auch zwischen unvollständigen Spezifikationen und diese implementierenden Programmen eine einfache Implikationsbeziehung. Man kann also auch eine formale Programmherleitung verwenden, solange man die informalen Anteile rechtzeitig ins Spiel bringt.

Wenn wir zum Beispiel eine imperative Menge implementieren, geben wir vielleicht als formale Nachbedingung von `set.insert(x)` einfach nur `set.has(x)` an, mit dem informalen Hinweis, dass natürlich alle anderen Elemente auch noch drin sind. Wenn wir diese Datenstruktur mit einem Array `A` und einem Zähler `i` mit Repräsentationsinvariante $set = A[0, ..i]$ implementieren, folgt für die Implementierung von `insert` die Nachbedingung $\exists j. 0 \leq j < i \wedge x = A[j]$. Wir könnten also ein beliebiges Element von $A[0, ..i]$ auf `x` setzen. Da wir aber an die informale Randbedingung denken, rühren wir $A[0, ..i]$ nicht an und implementieren: `i := i+1 ; A[i-1] := x`.

5.3.1 Effizienz als Grundanliegen, Optimierung als Beiwerk

Wie Tony Hoare sehr schön in einem Artikel [20] dargelegt, ist im Prinzip jede formale Spezifikation bereits ausführbar — wenn auch mit sehr hoher Laufzeit oder sogar ohne Terminierungsgarantie. An Stelle des “Verfeinern heißt ausführbar machen” sollten wir also besser sagen “Verfeinern heißt effizient ausführbar machen”! Die Effizienz wird damit zum Grundanliegen, denn sie ist die einzige Eigenschaft eines Programms, die erst im Programmcode zum Ausdruck kommt, und noch nicht in der Spezifikation.

Andererseits kennt man aber auch die Warnung an Programmierer, sich nicht in andauernden Optimierungen zu verlieren. Leitmotiv ist der Ausspruch von Donald Knuth: “Vorzeitige Optimierung ist die Wurzel allen Übels.” Unter “vorzeitig” versteht man “vor einem Performanztest, der die schwachen Teile des Programms aufzeigt.” Wenn aber jetzt Programmieren im Prinzip nur Effizienzsteigerung ist, sollten wir dann all unsere Spezifikationen ausführen und nur die beim Performanztest schwachen Teile verfeinern? Das würde aber die Anforderungen an die Spezifikationsprache und den Aufwand der Programmierung erheblich steigern!

Es ist daher angebracht zwischen der Effizienzsteigerung durch Implementierung und Effizienzsteigerung durch Optimierung zu unterscheiden. Aber wie sollen wir die Grenze ziehen? Eine wirklich absolute Grenze gib es nicht. Wie effizient man vor dem ersten Performanztest programmieren sollte, hängt vom Anwendungsbereich ab. Es gibt aber einige starke Heuristiken:

1. Zunächst erzielt die (Erst-)Implementierung generell eher qualitative Fortschritte, die sich meist in den asymptotischen Maßen der Komplexitätstheorie niederschlagen. Nachträgliche Optimierung hingegen erzielt “nur” bessere konstante Faktoren.
2. Die Implementierung einer Spezifikation ist relativ unabhängig von der zu Grunde liegenden Hardware (oder allgemeiner: der Laufzeitumgebung, auch “Plattform” genannt), während “Optimierung” oft auch bedeutet: Spezialisierung auf eine bestimmte Plattform.
3. Letztlich ist die Optimierung oft auch eine Spezialisierung unter Ausnutzung bestimmter Eigenschaften der Problemstellung, während “einfach implementierte” Programme eine allgemeinere Problemstellung lösen.

In den verschiedenen Anwendungsbereichen der Softwaretechnik gibt es jeweils Richtlinien dafür was nur “normal effizient” ist und was bereits “optimiert”. Dies kann sehr unterschiedlich sein und manchmal liegen diese Auffassungen sicher auch daneben, aber ohne solche Klischees müsste sich ein Programmierer ja ständig die Frage stellen “optimiere ich schon oder implementiere ich noch?”

Bei einem Sortieralgorithmus wird man in der Regel davon ausgehen, dass ein Programm mit asymptotischer Laufzeit $O(N \times \log(N))$ eine ausreichende Implementierung ist (besser geht's ja nicht), und jede weitere Verbesserung um eine Konstante ist Optimierung. Dies gilt insbesondere in Programmiersprachen, in denen ein $O(N \times \log(N))$ Programm genauso leicht zu schreiben ist wie ein $O(N^2)$ Programm. In imperativen Sprachen ist QuickSort der Algorithmus der Wahl, aber weil seine übliche in situ Implementierung etwas knifflig ist, mag sich manch einer mit einem einfacheren Sortieren durch Einfügen (oder Auswählen) zufrieden geben, dessen $O(N^2)$ Laufzeit ja für "kleine" Datensätze völlig ausreicht. Dann ist eben schon der Schritt zu QuickSort eine Optimierung.

Eine wichtige Konsequenz aus der Operationalisierungshierarchie ist, dass wir auch mehrere Verfeinerungsstufen haben können, die ausführbare Programme sind. Eine Optimierung ist wie ein weiterer Verfeinerungsschritt; das ursprüngliche Programm ist dann die Spezifikation des optimierten Programms. Falls man eine Optimierung durchführt, sollte man sich zu Nutze machen, dass man mit dem Urprogramm eine vollständige formale und sogar ausführbare Spezifikation zur Verfügung hat. Falls man nicht bereits eine überprüfbare Spezifikation hat (oft ist diese ja unvollständig), sollte man das Urprogramm auf jeden Fall als Dokumentation und als Generator für Test-Ergebnisse behalten. Selbst wenn man beides schon hat, kann das Programm vor der Optimierung eine gute Dokumentation des Algorithmus und der Verfeinerung sein.

Bei der ersten Implementierung einer Spezifikation entwirft man das Programm so einfach wie möglich. Was man dabei noch für "möglich" hält (also nicht von vornherein zu ineffizient), wird durch Erfahrungen und Klischees bestimmt (die zum größten Teil nicht von einem Anwendungsbereich auf einen anderen übertragbar sind). Manchmal führt man auch spezielle Vorstudien durch, um herauszufinden, ob eine bestimmte Implementierungsweise effizient genug ist. Wenn dann ein erstes lauffähiges Programm steht, wird man durch gezielte Messungen nur noch diejenigen Programmteile verändern, die sich als wahre Flaschenhälse erweisen. Dabei ist zu beachten:

1. Optimierungen machen das Programm komplizierter, dadurch erschweren sie die Fehlersuche (oder erhöhen die Fehlerwahrscheinlichkeit), und sie erschweren spätere Wartung.
2. Man sollte sich bewusst machen, welche Eigenschaften der Plattform und/oder der Problemstellung man für die Optimierung ausnutzt, und welche Einschränkungen bei der Wiederverwendbarkeit dadurch bedingt sind.

Kleiner Vorgriff auf die Modularisierung im nächsten Kapitel: Bei der Bewertung einer Optimierung ist ganz wesentlich, ob diese auch Schnittstellen ändert, oder ob man nur einige "versteckte" Implementierungen ändert. In ersterem Fall verkompliziert man das Programm wesentlich mehr, während man in letzterem Fall die Schnittstelle ja sogar noch aufwertet: Schnittstellen sind hauptsächlich *vereinfachte* Sichten auf eine (Modul-)Implementierung und damit natürlich umso wertvoller, je größer die "Komplexitätsdifferenz" zwischen Schnittstelle und Implementierung ist. Leider erfordern aber viele Optimierungen gerade ein "Transparent-Machen" von Schnittstellen, so dass deren Nutzer "direkt auf die unteren Schichten zugreifen" können, wie man so sagt. Deswegen kommt dem Entwurf von Schnittstellen eine so entscheidende Rolle zu, die in den folgenden Kapiteln behandelt wird.

6 Programmbausteine

Wenn man eine allgemein gültige Programmiermethode entwickelt, dann darf man sich natürlich nicht in den Details der verschiedenen Programmiersprachen verstricken. Dabei kommt zwangsläufig die Frage auf, welche Konzepte in der Programmierung die wichtigen sind, die überall wieder auftauchen. Und: wie stehen diese Konzepte zueinander, wie wendet man sie richtig an. Bei diesem Vergleich wird es immer darum gehen, was man mit welchem Konstrukt einfacher machen kann, denn die Relationen zwischen den Konstrukten sind alle formale Äquivalenzen, das heißt verschiedene Formulierungen unterscheiden sich überhaupt nur in ihrer Komplexität für verschiedene Aufgaben (und manchmal auch in der Effizienz).

Die folgende Liste folgt im Wesentlichen Bertrand Meyer [29]. Der Hauptunterschied besteht in der Trennung von funktionalen und imperativen Aspekten, die hier stark herausgearbeitet wird.

Ausdrücke und Funktionen Bausteine aller Programme.

strukturiertes Programmieren Keine Sprünge, viele Routinen.

strenge Typisierung und Generizität Offensichtliche Fehler gar nicht erst ins Programm lassen.

Modularität und Information Hiding Schnittstellen sind wesentlich einfacher als die Implementierungen.

Objekt-Orientierung Module (Klassen) können beliebig oft instanziiert werden; die Klassen definieren die Typen; alle Typen sind abstrakte Datentypen.

Vererbung Module können Spezifikation und Implementierung von anderen Modulen übernehmen.

Polymorphismus, Subtyping und dynamische Bindung Eine Variable kann Objekte verschiedenen Typs enthalten, die aber alle einen gemeinsamen Supertyp haben. Jeder Subtyp implementiert die Schnittstelle des Supertyps und verfeinert sie nach Bedarf. Alle Aufrufe sind implizite Fallunterscheidungen.

Notfallbehandlung Das Verhalten bei Versagen eines Teils der Software wird als separate Angelegenheit und getrennt vom normalen Programmfluss behandelt.

Die funktionale bzw. strukturiert imperative Programmierung liefert uns die Turing-vollständige Basis, die zur Erfüllung aller funktionalen Anforderungen ausreicht (man kann jede Spezifikation implementieren). Alle anderen Konzepte machen "nur" noch die Programmierung einfacher, nämlich durch Beseitigung negativer Redundanz und Anreicherung mit positiver Redundanz. Es geht immer wieder um Aufteilung und dann um Abgrenzung der Teile voneinander. Typen sind eine Art automatischer Abgrenzung (man muss sie sowieso hinschreiben) und bekommt den Konsistenz-Test dann kostenlos und vollverbindlich. Verträge hingegen muss man zusätzlich notieren, die Sprache dazu ist viel mächtiger, also auch komplizierter und leider ist der Konsistenz-Test nicht vollständig; er hängt von den Test-Eingaben ab.

Die Module trennen wir dabei nicht in ihre Aspekte *Schnittstellen*, *Instanzen*, *Vererbung*, *Polymorphismus*, sondern wir unterscheiden algebraische und imperative Module. Diese Unterscheidung ist analog zu Ausdruck/Befehl (engl. *expression/instruction*) und Funktion/Prozedur. Aber während bei diesen die Unterschiede fast vernachlässigbar sind, weil man sie leicht ineinander übersetzen kann, führen imperative Module neue Elemente ein, insbesondere wenn *Referenzen* mit ins Spiel kommen.

Im Folgenden werden wir uns diese Bausteine der Reihe nach genauer ansehen.

6.1 Ausdrücke und Funktionen

Am Anfang der Programmierung nahm man an, dass Variablen und Anweisungen ihre einfachsten Bestandteile sind. Die meisten Programmiersprachen enthalten Ausdrücke als einen ganz selbstverständlichen Bestandteil, der in immer gleicher Form auftaucht und daher oft gar nicht beachtet wird. (In vielen Handbüchern und selbst in den formalen Ansätzen Hoares Axiomatischer Semantik und Dijkstras Guarded Commands werden die Ausdrücke überhaupt nicht erwähnt. Sie sind einfach da.) Dabei spielen die Ausdrücke in theoretischer *und* praktischer Hinsicht eine wichtige Rolle. In den ersten Programmiersprachen spielte die Übersetzung von Ausdrücken (wenn auch nur numerischen) die Hauptrolle. Der Name FORTRAN deutet ja schon darauf hin, denn das Akronym bedeutet *formula translator*. Heute würde man “Ausdruck” statt “Formel” sagen.

Die primitiven Konstrukte der Programmierung mit Ausdrücken sind die Funktionsanwendung und die Funktionsdefinition. In den meisten Programmiersprachen wird mit der Funktionsdefinition gleich ein Name für die Funktion vergeben. Wir bleiben zunächst bei Funktionen, deren Rumpf nur aus einem Ausdruck besteht. In EIFFEL sieht das so aus:

```
some_function(argument : TYPE) : RESULT_TYPE is
do
  Result := expression
end
```

Die Vergabe eines Namens und die Definition einer Funktion sind eigentlich unabhängige Konzepte und in manchen Sprachen (wie zum Beispiel HASKELL) ist das auch so realisiert. Dort kann man Funktionen ohne Namen definieren (innerhalb von Ausdrücken ist das manchmal nützlich) und man kann auch Namen für Ausdrücke vergeben, die keine Funktionen sind. (In imperativen Sprachen nimmt man für letzteres einfach Variablen, an die man nur ein einziges Mal zuweist.)

Moderne Programmiersprachen machen sich auch zu Nutze, dass Operatoren (wie $+$ $-$ $*$ $/$) auch nur Funktionen sind, und behandeln sie mit den gleichen Regeln. Damit verbinden sie die Einfachheit von Ausdrücken, wie sie schon aus der Schule bekannt sind, mit der fundamentalen Mächtigkeit von Ausdrücken, die ja Grundlage *jeder* formalen Sprache sind. Die Prinzipien, die der funktionalen Programmierung zu Grunde liegen, braucht man immer, egal in welchem Paradigma man programmiert. Zum Beispiel gibt es in jeder Programmiersprache Regeln zum Gültigkeitsbereich (*scope*) von Namen, und solche lokalen Gültigkeitsbereiche sind gerade, was eine (mathematische) Funktion ausmacht. Auch die logischen Quantoren (\forall und \exists) definieren Bereiche. Und immer ist deren Semantik definiert durch Ersetzungsregeln und Umbenennungsregeln. Und das “Einsetzen” eines Argumentes in eine Funktion ist ja gerade diese Operation des Ersetzens. (Diese Beobachtung wurde nicht zuletzt von Hehner gemacht [14].) Stetiges Einsetzen von Argumenten in Funktionen ist übrigens genau die Art, wie funktionale Programme ausgeführt werden (zumindest in der üblichen “strikten” Semantik). Man kann also allein mit Funktionsanwendung und Funktionsdefinition bereits alle Programme schreiben, die man auch mit komplizierteren Programmiersprachen schreiben kann. Daher sollte man immer bewusst zwischen einem funktionalen und einem imperativen Stil auswählen. Man kann sich dabei an folgenden Vor- und Nachteilen orientieren:

- Die funktionale Programmiersprache verwendet dieselbe Notation wie ihre Semantik. Man braucht also keine speziellen Konstrukte (wie Schleifen, Sequenzen), die nur der Programmierung dienen und nicht hauptsächlich der Spezifikation.
- Viele Probleme und noch mehr Teilprobleme von größeren Problemen lassen sich auf natürlichste Weise als Funktionen auffassen.
- Durch die Freiheit von Seiteneffekten ist das Rechnen mit Funktionen und Ausdrücken wesentlich einfacher und entspricht im wesentlichen der jedem bekannten Arithmetik.

- Durch die Unabhängigkeit von Zuständen des Computers wird oft ein höheres Abstraktionsniveau erreicht. Das macht Programmteile wiederverwendbarer.
- Da funktionale Objekte stets unveränderlich sind, haben Werte und Referenzen stets dieselbe Semantik. Man kann also das Programm unabhängig davon entwickeln, ob Objekte selbst kopiert werden oder nur Referenzen darauf. Man kann diese Entscheidung sogar dem Compiler überlassen, ohne dazu die Programmiersprache ändern zu müssen!

Und hier die Vorteile imperativer Programmierung:

- Viele Probleme haben eine ganz natürliche Beziehung zu Zuständen, die in imperativen Programmen explizit beschrieben werden.
- Auch Computer basieren intern auf Zuständen. Durch den direkten Zugriff darauf haben imperative Programme einen Effizienz-Vorteil.

“Direkt” heißt, dass man keine speziellen Theorien und optimierenden Compiler braucht, wie das bei imperativen Funktionalitäten funktionaler Programmiersprachen der Fall ist.

Arithmetische Operationen erledigt man ja selbst in imperativen Sprachen auf funktionale Art und Weise: $s := s*s + 1$. Dessen imperative Version würde auf das Register-Niveau der Maschinensprachen zurückfallen: `s.mult(s)` ; `s.add(1)`. Aber leider sind die meisten Programmierstile seit den Zeiten von FORTRAN noch nicht weiter gekommen: Arithmetik (und manchmal die Verarbeitung von Zeichenketten) ist immer noch das einzige Gebiet, das man funktional behandelt. Aber dabei gibt es doch so viele weitere Gebiete, die völlig zustandslos existieren, und bei abstrakten Datentypen ist man sowieso vom Niveau des Computers so weit entfernt, das auch die Effizienz nicht so sehr ins Gewicht fällt.

Als Beispiel dazu können wir uns eine Bibliothek zur Erstellung von HTML-Dateien anschauen. Eine solche Bibliothek ist nützlich, wenn man Web-Seiten dynamisch generieren will, denn dann werden die Seiten ja im Programmcode beschrieben. Für die Sprache PERL gibt es eine solche Bibliothek in imperativer Geschmacksrichtung, und für HASKELL in funktionaler Tönung. Zum leichteren Vergleich können wir uns beide Varianten in EIFFEL vorstellen. Wo ist der Unterschied in der Verwendung? Ganz einfach: in der imperativen Variante erstellt man ein HTML-Objekt und fügt dann Elemente hinzu; in der funktionalen Variante kombiniert man Elemente zu HTML-Objekten, wobei Elemente und Objekte auf der gleichen Stufe stehen. Die imperative Variante wird möglicherweise versuchen besonders effizient zu sein, und die HTML-Daten auszugeben (in eine Datei oder an das Netzwerk) noch während sie generiert werden. Wenn d ein imperatives Dokument ist, wird dann der Aufruf `d.append(element)` das *element* direkt umwandeln und ausgeben. Im Gegensatz dazu ist die funktionale Variante flexibler: man kann mit derselben Routine auch ein Element vorne anhängen, indem man schreibt: `d := element.append(d)`. Dies macht es leichter, Angelegenheiten zu trennen, und man erhält eine wesentlich einfachere Bibliothek, weil man viel weniger Routinen braucht. Und das alles ohne die Programmiersprache zu wechseln!

Viele schöne Beispiele für die Manipulation funktionaler Programme findet man in [2]. An Stelle diese hier wiederzugeben, komme ich im Folgenden noch einmal auf die Kombination imperativer und funktionaler Aspekte zurück.

6.2 Einfache Seiteneffekte

Beim Programmieren im Kleinen sind elementare Konstrukte mit Seiteneffekten die Zuweisung und der Prozedur-Aufruf. In einer “puren” Programmiersprache (ohne Systemaufrufe), in der

man alle Prozeduren selbst schreiben muss, ist die Zuweisung überhaupt das einzige Konstrukt, dass Seiteneffekte erzeugen kann. Erst wenn wir Zuweisungen haben, brauchen wir überhaupt die üblichen Kontrollstrukturen Sequenz, Fallunterscheidung und Schleife!

Fallunterscheidungen sind dabei besonders einfach zu verstehen, wenn man sie unter dem Aspekt des Aufteilen-Zusammenführen betrachtet: Eine Problemstellung wird in zwei (oder mehr) Fälle aufgeteilt, diese werden separat gelöst und beide Programnteile tragen zum selben Gesamt-Ergebnis bei. Auch wenn die übliche Form der Fallunterscheidung unsymmetrisch ist (wegen des `else` Teils, der sozusagen alle restlichen Fälle abdeckt), ist es hilfreich sie sich als symmetrisch vorzustellen: in jedem der beiden Fälle haben wir etwas gegeben, das wir in dem anderen Fall nicht haben. Wenn wir zum Beispiel einen Fall unterscheiden, indem eine Menge leer ist, so gibt es im anderen Fall mindestens ein Element, das wir benutzen können. Damit verschafft uns die Fallunterscheidung die notwendige Vorbedingung für den Zugriff auf ein Element. Genauso kann uns eine Fallunterscheidung vor einer Division durch Null beschützen oder vor dem Zugriff auf leere Referenzen. Auf diese Weise ist sofort ersichtlich, dass Vorbedingungen eingehalten werden und wirklich alle Fälle abgedeckt sind. (Natürlich sind die Dinge nicht immer so einfach, sonst würden wir ja keine Zusicherungen brauchen.)

Besonders interessant ist auch der Zusammenhang zwischen imperativer und funktionaler Fallunterscheidung. APTOPenthält dazu folgendes Lemma:

```
if a then x := b else x := c
= x := if a then b else c
```

Leider sind aber die meisten Programmiersprachen entweder imperativ oder funktional und kennen dementsprechend nur eine der beiden Varianten der Fallunterscheidung. Man kann das aber trotzdem noch benutzen, um zwischen verschiedenen Sprachen zu übersetzen. In imperativen Sprachen kann man auch die funktionale Fallunterscheidung in Form ihrer logischen Definition verwenden; zum Beispiel in EIFFEL: `a implies b and not a implies c`.

6.2.1 Ein Schleifenidiom

Die Regeln für Schleifen sind eigentlich in [34] schon gut genug erklärt. Wir wollen uns hier nur ein häufiges Muster anschauen, das die Verwendung von Schleifen in “richtigem Code” illustriert. Die generelle Form einer Schleife in EIFFEL ist:

```
from Initialisierung
invariant Invariante I
variant Zähler-Schranke
until Abschluss-Bedingung C
loop Schleifenkörper
end
```

Wir können uns vorstellen, dass die Ausdrücke I und C den Vertrag der Schleife ausmachen. Sowohl den Vertrag mit sich selbst als auch den Vertrag mit ihrer Umgebung.

Der Aspekt “mit sich selbst” ist dabei besonders wichtig, denn er erlaubt uns unbeschränkt viele Durchläufe der Schleife zu verstehen und dabei nur beschränkt viel nachzudenken. Die Invariante beschreibt gleichzeitig alle Zwischenzustände der Schleife. Wenn wir die Initialisierung und den Schleifenkörper so programmieren, dass er die Invariante erfüllt, haben wir alle Fälle abgedeckt inklusive den ersten und den letzten, mit denen allein sich Amateurprogrammierer schon ewig ablagen. Die Regel “Achten Sie besonders auf Sonderfälle” gilt eben hier nicht, wenn man methodisch vorgeht. Voraussetzung für diese Einfachheit ist übrigens EIFFELS Prinzip, dass Schleifen jeweils nur einen Eingang und einen Ausgang haben, man kann nicht beliebig hinein und heraus springen, sondern muss immer an der Invariante vorbei.

In diesem Abschnitt interessiert uns besonders die Schnittstelle zwischen einer Schleife und dem nachfolgenden Code. Nach der Schleife gilt ja die Zusicherung $I \wedge C$. Wir wollen dazu ein häufiges Muster betrachten:

```

until A or B
loop ...
end

if A then ...
else check B end ...
end

```

Bei einer Abbruchbedingung $A \vee B$ ist die Nachbedingung der Schleife folglich $I \wedge (A \vee B)$ also gleich $(I \wedge A) \vee (I \wedge B)$. Im Muster wird diese Bedingung weiterverarbeitet, indem man jeden der Fälle getrennt behandelt und zwar meist einen mit Vorrang (im Beispiel Fall A). Die `check`-Bedingung ist übrigens schwächer als nötig — an dieser Stelle gilt $\neg A \wedge B$, nur braucht man das meist nicht. So sieht das Muster aus, wenn man es auf typische Such-Probleme (zum Beispiel in Datenstrukturen) verwendet:

```

from
  gehe_zum_Anfang
invariant
  bisher_nicht_gefunden
until
  gefunden or am_Ende
loop
  gehe_weiter
end

if gefunden
then
  ...
else
  check gibt_keins end
  ...
end

```

Der Platzhalter *gefunden* bedeutet hier, dass das Element in einer lokalen Variable verfügbar ist (zum Beispiel durch einen Iterator). Daraus folgt, dass die Bedingungen *gefunden* und *gibt_keins* ausschließlich sind: es gilt $\neg(\textit{gefunden} \wedge \textit{gibt_keins})$ und folglich können wir die Fallunterscheidung auch anders herum aufschreiben, je nachdem was leichter zu lesen ist.

Dieses Beispiel ist auch deshalb besonders interessant, weil herkömmlicherweise für dieses Muster keine reinen Schleifen eingesetzt werden: man verwendet `goto`-ähnliche Konstrukte — in den meisten Sprachen heißt das `break`. Diese Konstrukte machen zwar den Kontrollfluss explizit, aber wie man hier sehen kann, macht es die Einschränkung der Schleifen auf einen Ein- und Ausgang leichter zu sehen, was die Schleife und ihr umgebender Code im Ganzen tun.

6.2.2 Prozeduren

Für den Theoretiker ist das Konzept der Prozedur ein “Pauschalangebot”, das drei Dinge in sich vereint: Unabhängige Verfeinerung, Parameter und lokale Variablen. Unabhängige Verfeinerung bedeutet, dass jede Implementierung, die die Spezifikation erfüllt, auch automatisch den Zweck des Aufrufers erfüllt. Der Aufrufer kann so tun, als bestünde die Prozedur nur aus ihrer Spezifikation, und Implementierungen kann man erstellen, ohne den oder die Aufrufer überhaupt zu kennen. Dies ist der wesentliche Abstraktionsgewinn durch Prozeduren.

Lokale Variablen sind ein prinzipiell von Prozeduren unabhängiges Konzept (in manchen Sprachen sind sie auch unabhängig von diesen realisiert), aber in modernen Sprachen sind sie an Routinen gekoppelt — genau wie in der funktionalen Programmierung Gültigkeitsbereiche an

Funktionen gekoppelt sind. Diese Kopplung ist methodisch allein schon deswegen von Vorteil, weil Programmierer so ermuntert werden, immer eine neue Routine zu schreiben, wenn eine neue Variable gebraucht wird. Dadurch bleiben Routinen immer schön kurz und übersichtlich. Dies ist insbesondere dann nützlich, wenn ein Programm im Prinzip nur Anforderungen formalisiert, also selbst wenig informatische Algorithmik enthält.

Es folgt ein Beispiel für so eine Routine. Dort haben wir nicht einmal lokale Variablen: die Funktion ist einfach nur eine Abkürzung für den enthaltenen Ausdruck.

```
integer_square_root( i : INTEGER ) : INTEGER is
  require
    i >= 0
  do -- ensure
    Result := i.to_real.square_root.floor
  end
```

Die Vorbedingung $i > 0$ wird einfach nur für die Funktion `square_root` weitergegeben; die Nachbedingung besteht im Routinenkörper selbst: die Gleichung

```
Result = i.to_real.square_root.floor
```

explizit aufzuschreiben, würde nur zu viel Redundanz erzeugen, deshalb verwenden wir das Kommentarmuster `do -- ensure`; damit wird die Gleichung automatisch in der Klassenschnittstelle als Nachbedingung aufgenommen.

Eine solche Verwendung von Routinen entspricht ungefähr dem Zweck von Definitionen in der Mathematik: man spart zwar nicht wirklich viel Tipparbeit, aber dafür hat man auch die Garantie, dass man jedes Mal genau dieselbe Definition verwendet. Außerdem erhält man dadurch die Möglichkeit Aussagen zusammenzufassen, die man über den definierten Begriff gelernt hat. Wenn die obige Funktion zum Beispiel in einem Programm verwendet wird, dass viel mit Ganzzahlen operiert, wird man leicht auf folgende Aussagen stoßen:

```
integer_square_root( i : INTEGER ) : INTEGER is
  require
    i >= 0
  do -- ensure
    Result := i.to_real.square_root.floor
  ensure
    -- alternative definition:
    Result * Result <= i
    i < (Result+1) * (Result+1)
  end
```

Wir haben hier eine Nachbedingung hinzugefügt, die mit der Funktionsdefinition äquivalent ist (wovon man sich leicht durch Arithmetik und die Eigenschaften der `floor`-Funktion überzeugen kann). Sowohl Kunde als auch Implementierer der Routine können sich jeweils eine der beiden alternativen Definitionen aussuchen und sie verwenden, während die automatische Prüfung der Zusicherungen sicherstellt, dass die Definition in der Nachbedingung wirklich von der anderen impliziert wird. (Die umgekehrte Implikation wird nicht überprüft, siehe dazu im nächsten Abschnitt.)

Routinen als Definitionen sind also sehr nützlich (ganz besonders, wenn der “Programmieren als Formalisieren”-Aspekt überwiegt), aber im nächsten Abschnitt sehen wir, dass ein noch viel größerer Nutzen erzielt wird, wenn Routinen nicht mehr ihre eigene Spezifikation darstellen; dann kommen wir mit Hilfe von Zusicherungen zu einem Fortschritt, den wir ohne sie nie erreichen könnten.

6.2.3 Routinierte Abstraktion

Als C- oder C++-Programmierer kannte ich bei Programmtests zwei Arten negativer Ergebnisse: entweder das Programm stürzt ab mit der lapidaren Meldung “Speicherschutzverletzung”

oder “Ungültiger Maschinenbefehl”, oder eines der Testergebnisse ist falsch. Bei vollständiger Verwendung von Zusicherungen in EIFFEL hingegen erhielt ich stets einen Programmabsturz mit Angabe der fehlgeschlagenen Zusicherung inklusive aller Variablenbelegungen aller aktiven Routinen. Wie ein großer Finger, der auf die fehlerhafte Programmstelle zeigt! Dieser Vergleich zeigt auf überspitzte Weise den Vorteil der Methode. Natürlich können auch die Werkzeuge von C und C++ Absturzstellen und Variablenwerte anzeigen. Aber nur durch konsequent eingesetzte Zusicherungen liegt die Absturzstelle wirklich nah bei der fehlerhaften Programmstelle.

Im Extremen Programmieren ist die Redundanzfreiheit eine der wichtigsten Qualitätsmerkmale von Programmcode; sie erlaube es, Programme leicht weiterzuentwickeln und Fehler leichter zu finden und zu beseitigen. Diese Konzentration auf die Abwesenheit negativer Redundanz setzt aber voraus, dass der Aspekt “Formalisieren” in der Programmierung überwiegt. Der Grund ist ganz einfach, dass das Operationalisieren immer negative Redundanz einfügen wird, und man braucht positive Redundanz, um deren Konsistenz zu sichern: Schleifeninvarianten sichern zum Beispiel die Konsistenz zwischen Initialisierung und Schleifenkörper; Klasseninvarianten braucht man allein schon, um die Konsistenz von bidirektionalen Referenzen zwischen Klassen zu sichern (Beispiel im Abschnitt 7.3.5). Außerdem ist algorithmischer Code nicht mehr seine eigene Spezifikation, und wir brauchen die Zusicherungen der positiven Redundanz auch als Dokumentation.

Zu XP nennt man immer als Voraussetzungen für die erfolgreiche Anwendung, dass die Anforderung erst während des Projekts entstehen, dass das Team nicht zu groß sein darf, und andere mehr. Für viele der XP-Techniken, insbesondere den Verzicht auf jede Art von Redundanz und Dokumentation (außer dem Quellcode selbst), ist aber auch Voraussetzung, dass die Programme wirklich ausführbare Spezifikationen sein können, also dass der Aspekt des Formalisierens in der Programmierung überwiegt. Sobald algorithmische oder andere technische Angelegenheiten eine Rolle spielen, sinkt die Effektivität von XP drastisch, weil die notwendige positive Redundanz in XP einfach ignoriert wird.

Die wichtigsten Zusicherungen in Programmen sind die Spezifikationen von Routinen: sie sind wichtig, um die Routinen zu verstehen und gleichzeitig leicht lesbar. Außerdem sind sie die Bausteine, die die komplizierten Modulspezifikationen bilden. Die Spezifikation einer EIFFEL-Routine `require P ... ensure Q` entspricht der prädikativen Spezifikation $P \Rightarrow t < \infty \wedge Q$. Wie wir im Abschnitt über die formalen Theorien gesehen haben, “beschützt” die Vorbedingung hier nicht nur die Nachbedingung, sondern auch die Terminierungsaussage. Eine Routine kann also die Gültigkeit der Vorbedingung immer voraussetzen und bei Nichteinhaltung sogar abstürzen — das vereinfacht die Programmierung erheblich.

Wir können uns jetzt anschauen, was mit unserem Beispiel aus dem vorigen Abschnitt passiert, wenn wir den spezifikativen Routinenkörper durch einen “richtigen Algorithmus” ersetzen. (Man stelle sich vor, die verwendete Fließkommafunktionalität steht auf der Zielplattform nicht zur Verfügung oder ist zu langsam. Das Beispiel stammt übrigens aus [43], wo es noch etwas ausführlicher behandelt wird.)

```
integer_square_root( i : INTEGER ) : INTEGER is
  require
    i >= 0
  local
    j : INTEGER
  do
    from
      j := i
    invariant
      0 <= Result and Result * Result <= i      -- A
      1 <= j and j <= i                          -- B
    variant
      i - Result + j
```

```

    -- Either we decrease 'j' or we increase 'Result'.
until
  i < (Result + 1) * (Result + 1)      -- C
loop
  if
    i < (Result + j) * (Result + j)    -- D
  then
    check j > 1 end -- from "not C and D", i.e.
    -- (Result+1)^2 <= i < (Result+j)^2
    j := j \ 2
  else
    Result := Result + j
  end
end
ensure
  Result := i.to_real.square_root.floor
  -- alternative definition:
  Result * Result <= i                -- A
  i < (Result+1) * (Result+1)        -- C
end

```

Bei diesem kniffligen Algorithmus ist es sehr leicht, einen Fehler zu machen! Aber durch die Invariante kann man die Korrektheit (inklusive aller Spezialfälle!) leicht in mehreren kleinen Schritten überprüfen. Dabei ist fast noch am schwierigsten einzusehen, dass j niemals bis auf Null verringert werden kann (dann würde sich die Schleife endlos drehen), deswegen habe ich dazu in der Zeile vor j s Halbierung noch eine Zusicherung mit Erklärung eingebaut.

An dieser Implementierung der Routine können wir den wahren Vorteil prozeduraler Abstraktion erkennen: Die Schnittstelle der Funktion ist immer noch genau dieselbe, einfache wie im vorigen Abschnitt, obwohl die Implementierung jetzt viel komplizierter ist. Mit dieser Routine erhält ein Aufrufer nicht nur eine nützliche Definition, sondern wirklichen Mehrwert: er spart sich nämlich die komplizierte Implementierung und ihre Verifikation. Und weil die Schnittstelle wirklich alle notwendigen Informationen enthält, muss er sich die Implementierung auch niemals anschauen. In diesem Fall entspricht der Nutzen der Routine eher einem mathematischen Lemma (oder Theorem), dessen Beweis man ein einziges Mal erstellt, um sich dann später je nach Belieben und Bedarf darauf zu berufen!

Auch die Überprüfung der Nachbedingung ist bei so einer Implementierung unerlässlich: wir können uns nicht mehr auf die XP-Methode der Redundanzbeseitigung verlassen, um Fehler zu finden, weil die Redundanz inhärent im Algorithmus ist. Und noch ein Vorteil der kombinierten objektorientiert-funktionalen Methode: obwohl die Funktion imperativ implementiert ist, kann sie der Aufrufer ganz wie eine mathematische Funktion verwenden, so als würde er an Stelle von `i.integer_square_root` schreiben: `i.to_real.square_root.floor`.

Aber auch wenn man Routinen informal spezifiziert, sollte man sich die Abstraktion zu Nutze machen, damit lokale Variablen nicht nur syntaktisch in den Prozeduren gekapselt sind, sondern auch semantisch. Beispiel: “[Die Prozedur] merkt sich den aktuellen Zustand des Systems, führt eine Komplett-Überprüfung durch, und stellt den aktuellen Zustand wieder her.” In dieser Beschreibung kommt quasi eine anonyme Variable vor, die den zwischengespeicherten Zustand enthält. Besser wäre die Beschreibung per Nachbedingung: “System überprüft, Zustand nicht verändert.”

Ob man Routinen nun in Form von Definitionen oder Lemmas verwendet oder in einer Zwischenform (man denke daran: Formalisieren und Operationalisieren sind verwundene Prozesse), in jedem Fall sind sie ein reines Struktur-Element des Codes, dienen also ganz allein dem besseren Verständnis und der Modularisierung. Diese Exklusivität kommt zustande weil Compiler je

nach Effizienz einen Routinenaufruf durch einen Sprung oder den Routinenkörper übersetzen. In diesem Sinne war es einer der wichtigsten Beiträge der strukturierten Programmierung, den großzügigen Einsatz von Routinen zu empfehlen. (Damit erledigte sich übrigens auch gleichzeitig die damals so umstrittene `goto`-Problematik, denn `gotos` verlieren ihren Schrecken, wenn ihre Sprungpotenz auf die nunmehr sehr kleinen Routinen beschränkt wird.)

Die Routinen sind also das erste nicht-algorithmische Konstrukt, das den Lösungsraum aufbläht, wie in Abschnitt 5.1.3 beschrieben. Wir werden im Abschnitt 7.3 sehen, dass der Umgang mit Routinen auch sehr einfach ist: wenn man die Techniken kennt, findet man leicht zur einer einfachsten Lösung.

6.2.4 Die Vorteile von Vorbedingungen

In der “Programmieren ist Spezifizieren”-Methode des Extremen Programmierens sieht man Vorbedingungen als Fehlerquellen in der Software an und empfiehlt sie zu vermeiden. Die Verfechter dieser Ansicht sehen in jeder Vorbedingung eine Möglichkeit diese zu verletzen und damit einen potentiellen Programmabsturz. Ihre Entwurfsregel lautet: “Man überlege sich, was der Aufrufer einer Routine wohl in dem Fall machen würde, den man mit der Vorbedingung ausschließt, und dann lässt man genau dies von der Routine machen und entfernt dafür die Vorbedingung.” Dieses Verfahren ist natürlich logisch korrekt und ist auch in manchen Fällen die richtige Entwurfs-Entscheidung; es aber als allgemeines Entwurfsprinzip hinzustellen, ist absolut kontra-produktiv: man muss sich nur überlegen, dass eine Routine ja eigentlich eine Abstraktion darstellt, die von mehreren Aufrufern verwendet wird. All diese Aufrufer haben gemeinsam, dass sie die Routine nur aufrufen, wenn ihre Vorbedingung erfüllt ist, aber was sie im anderen Fall tun, ist natürlich unterschiedlich! Wenn man in so einem Fall die Routine erweitert, um die Vorbedingung zu entfernen, würde sie die Komplexität aller ihrer Aufrufer in sich zusammen fassen; von Modularität bleibt dann nichts mehr übrig.

Natürlich sind *totale Funktionen* (solche ohne Vorbedingungen) oftmals sehr praktisch, aber ein guter Entwurf kann ohne *partielle Funktionen* (mit Vorbedingungen) nicht auskommen. Und dabei muss man keine Angst haben, dass verletzte Vorbedingungen eine stetige Quelle von Fehlern sind. Denn wenn die Vorbedingung in der Klassenschnittstelle dokumentiert ist, wird sie jeder Programmierer sehen, der die Routine aufruft. Und wenn sich doch mal ein Fehler einschleicht, so lässt diesen die automatische Überprüfung der Zusicherungen rasch auffliegen. DESIGN BY CONTRACT gibt dem Programmierer also die Freiheit, totale oder partielle Funktionen zu verwenden je nachdem, was dem einfachen Entwurf zuträglicher ist, und stets ohne das Risiko dabei Fehlerquellen einzuführen. Natürlich sind Routinen ohne Vorbedingungen leichter zu verwenden, insbesondere Ausdrücke lassen sich leichter umformen, wenn alle verwendeten Funktionen (insbesondere die Operatoren!) total sind. Eine schwächere (oder keine) Vorbedingung macht natürlich immer einen Vertrag zugunsten des Aufrufers (des Kunden) aus. Aber einfacher Entwurf zeichnet sich ja gerade dadurch aus, dass nur soviel geliefert wird, wie auch wirklich benötigt, denn mit einer (stärkeren) Vorbedingung wird natürlich prinzipiell die Implementation leichter.

Hier also die Vorteile von Vorbedingungen:

1. Vorbedingungen erlauben es Zusicherungen von einer Routine zur nächsten weiter zu reichen. — Sonst müsste man jedes Mal wieder bei Null anfangen.
2. Wenn man eine Routine direkt testet, dann ist ihre Nachbedingung mit den Testfällen redundant — beide werden auf Konsistenz zueinander und zum Routinenrumpf geprüft. Die Vorbedingungen sichern aber die innere Überprüfung von Routinen und helfen dadurch (Aufruf-)Fehler zu lokalisieren, deren Existenz ein Nachbedingungstest nur aufzeigt.
3. Routinen mit Vorbedingung tun “weniger” und ihr Verhalten ist einfacher zu spezifizieren. So erhält man leichter Bausteine, die man zu größeren Gebilden zusammenfügen kann.

6.3 Fallstudie: Schrittweise Verbesserung einer großen Routine

Ein Stück Code ist schlecht, wenn es nicht funktioniert, oder wenn man es man noch viel besser machen kann. Diese ad-hoc Definition bezieht sich direkt auf die zwei Kernqualitäten Korrektheit und Einfachheit. Ein inkorrektes Programm erkennt man an einem Versagen, aber ein zu kompliziertes Programm erkennt man nicht so leicht. Dass das Programm *kompliziert* ist, kann ja auch an der inhärenten Komplexität des Problems liegen. Dass es *zu kompliziert* ist, lässt sich nur durch Angabe einer einfacheren Version zeigen. Ähnliches trifft übrigens auch auf die Effizienz zu: man kann zwar sehen, dass ein Programm langsam ist, aber deswegen weiß man noch lange nicht, ob es auch schneller geht. Um zu zeigen, dass ein Programm optimal-effizient ist, braucht man eine spezielle Argumentation, die sich eher am Problem orientiert als am Programm. Genauso bei der Einfachheit: um sich zu versichern, dass ein Programm optimal-einfach ist, muss man das Problem gut verstanden haben und muss Entwurfsentscheidungen anhand dieses Verständnisses begründen.

Das folgende Beispiel ist auch deshalb interessant, weil der schlechte Code (der in C geschrieben ist) zwei `return` Anweisungen enthält. Ich habe beim Programmieren in C dieses `return` oft praktisch gefunden: Wenn man an einer Stelle der Funktion die Nachbedingung erfüllt hat, macht man einfach ein `return` und dieser Fall ist erledigt — im Rest der Routine betrachtet man nur noch die anderen Fälle. Die formale Semantik von `return` ist also sehr einfach, man kann sich jetzt fragen, warum dieses Konstrukt in der *besten Programmiersprache der Welt* (EIFFEL) nicht enthalten ist. Gibt es Probleme, deren Lösung in EIFFEL dadurch komplizierter wird?

Schauen wir uns mal den Original-Quelltext an:

```

1 static void rsoc_sweep(rsoc*c) {
2     na_env*nae=c->nae;
3     rsoh*gp=(rsoh*)&(c->first_header);
4     rsoh*pp;
5     rsoh*eoc=((rsoh*)((char*)c)+c->header.size));
6     c->free_list_of_large=NULL;
7     if (c->header.size > RSOC_SIZE) {
8         if (gp->header.magic_flag == RSOH_MARKED) {
9             gp->header.magic_flag=RSOH_UNMARKED;
10            c->next=nae->chunk_list;
11            nae->chunk_list=c;
12        }
13        else {
14            c->header.state_type=RSO_FREE_CHUNK;
15        }
16        return;
17    }
18    while (gp<eoc) {
19        while (gp->header.magic_flag == RSOH_MARKED) {
20            gp->header.magic_flag=RSOH_UNMARKED;
21            gp=((rsoh*)((char*)gp)+gp->header.size));
22            if(gp>=eoc) {
23                c->next=nae->chunk_list;
24                nae->chunk_list=c;
25                return;
26            }
27        }
28        gp->header.magic_flag=RSOH_FREE;
29        pp=(rsoh*)((char*)gp)+gp->header.size);
30        while ((pp<eoc)&&(pp->header.magic_flag != RSOH_MARKED)) {
31            pp->header.magic_flag=RSOH_FREE;

```

```

32     gp->header.size+=pp->header.size;
33     pp=((rsoc*)((char*)pp)+pp->header.size));
34 }
35 if (gp->header.size >= RSOC_MIN_STORE) {
36     if (nae->store_left==0) {
37         nae->store_left=gp->header.size;
38         nae->store=gp;
39         nae->store_chunk=c;
40     }
41     else if (nae->store->header.size < gp->header.size) {
42         ((fll_rsoh*)nae->store)->nextfllol=nae->store_chunk->free_list_of_large;
43         nae->store_chunk->free_list_of_large=((fll_rsoh*)nae->store);
44         nae->store_left=gp->header.size;
45         nae->store=gp;
46         nae->store_chunk=c;
47     }
48     else {
49         ((fll_rsoh*)gp)->nextfllol=c->free_list_of_large;
50         c->free_list_of_large=((fll_rsoh*)gp);
51     }
52 }
53 gp=pp;
54 }
55 if (((rsoc*)&c->first_header)->header.size >=
56     (c->header.size-sizeof(rsoc)+sizeof(rsoc))){
57     c->header.state_type=RSO_FREE_CHUNK;
58     nae->store_chunk=NULL;
59     nae->store_left=0;
60 }
61 else{
62     c->next=nae->chunk_list;
63     nae->chunk_list=c;
64 }
65 }

```

Diese Prozedur verstößt bereits gegen die Regel, dass eine Routine nie länger als eine Bildschirmseite sein sollte. Mit ihren 65 Zeilen übertrifft sie selbst den größten Bildschirm. (Text-Fenster fassen i.d.R. 20 bis 40 Zeilen.) Ausserdem sind die unaussprechbaren Bezeichner ein klarer Regelverstoss. Aber ansonsten hat sich die strukturierte Programmierung bereits gut ausgewirkt: man sieht auch ohne Kommentare, wozu die lokalen Variablen gut sind; das heißt zunächst einmal, welches Laufvariablen sind und welche konstant bleiben. Die Routine enthält eine `return` Anweisung in Zeile 16 und eine in Zeile 25. Die erste entspricht einem klassischen Muster: Man handelt einen Spezialfall ab, und anstatt den ganzen Rest der Routine in einen `else`-Teil einzukapseln, springt man einfach aus der Routine heraus. Da die Routine in diesem Fall sowieso schon zu lang ist, kann man hier einfach den ganzen `else`-Teil in eine Unter-Routine auslagern — man löst damit zwei Probleme auf einmal. Ich habe ja schon genug darüber geschrieben, wie toll Routinen sind — und hier ist ein klassischer Anwendungsfall.

In folgendem Code-Beispiel habe ich die C-Routine einfach mal in drei Teile geteilt: `cond1`, `stuff1` und `real_array_sweep`. Der Lesbarkeit halber verwende ich EIFFEL-Syntax, die zweite Version ist bereits die geänderte:

```

1  array_sweep( chunk : ARRAY_CHUNK ) is
2  do

```

```

3         if cond1(chunk)
4         then
5             stuff1(chunk)
6             return
7         end
8         real_array_sweep(chunk)
9     end -- array_sweep
10
11 array_sweep2( chunk : ARRAY_CHUNK ) is
12     do
13         if cond1(chunk)
14         then stuff1(chunk)
15         else real_array_sweep(chunk)
16         end
17     end -- array_sweep2

```

Wenn man diese Routinen nach C übersetzt, so existieren für die drei Lücken `cond1`, `stuff1` und `real_sweep` eine (Bool'sche) Funktion und zwei Prozeduren, so dass das Programm noch exakt denselben Effekt hat, wie obiges Original. Nach dieser Abstraktion ist klar ersichtlich, dass `array_sweep2` semantisch äquivalent zu `array_sweep` ist. Ich habe die Routine `rsoc_sweep` übrigens unter Kenntnis des Routinenkontexts in `array_sweep` umbenannt. Ebenso erhalten alle anderen Bezeichner im Laufe dieser Diskussion anständige Namen, die dann auch klar werden, wenn wir langsam herausfinden, was die Routine eigentlich macht. Ich werde das allerdings nicht komplett erklären: Stellen wie `cond1` und `stuff1` lassen wir einfach unberührt, während wir bei `real_sweep` weiter graben. Schauen wir uns mal die abstrahierte Version an:

```

1     real_array_sweep( chunk : ARRAY_CHUNK ) is
2         local
3             head : ARRAY_HEADER
4         do
5             from head := chunk.first_header
6             until head = chunk.after_last_header
7             loop
8                 from
9                 until not head.is_marked
10                loop
11                    head.mark
12                    head := head.next_header
13                    if head = chunk.after_last_header
14                    then
15                        stuff2
16                        return
17                    end
18                end
19                collect_array_segment(chunk, head)
20                recycle_array_segment(chunk, head)
21                head := head.next_header
22            end -- loop over segments
23            if cond2
24            then stuff3
25            else stuff2
26            end
27        end -- real_array_sweep

```

Das zweite `return` in der schlechten schlechten C-Routine befindet sich in der abstrahierten Version direkt in der Mitte. Es ist dieses `return`, das meine Aufmerksamkeit auf dieses Beispiel gezogen hat. Hier springt der Programmierer nicht nur aus der Routine, sondern auch noch aus zwei Schleifen und einer Fallunterscheidung. Und was nun wirklich schlecht ist: die Fallunterscheidung in der Schleife testet genau die Abbruchbedingung! Redundant!

Kurzeinführung in den Programmkontext: Die Routine stammt aus einem *mark & sweep* Garbage Collector. Dieser hat eine *mark*-Phase, in der alle noch benötigten Objekte markiert werden, und eine *sweep*-Phase, in der alle unmarkierten Objekte gelöscht werden. Unsere Routine ist für das Löschen von Arrays verantwortlich. Jedes allokierte Array bildet zusammen mit einem *header*, der dessen Größe enthält, ein *segment*. Mehrere Segmente werden zusammen in einem *chunk* gespeichert, und die Routine kümmert sich jeweils um einen solchen *chunk*.

Die äußere Schleife läuft über alle *header/segments* eines *chunk*. Von jedem dieser Segmente wird die Markierung entfernt, falls es markiert ist, andernfalls wird das Segment der Freiliste hinzugefügt (via `recycle_array_segment`). Der Aufruf `collect_array_segment` vereinigt das Segment mit eventuell folgenden, ebenfalls unmarkierten Segmenten (diese hören dadurch auf als solche zu existieren und werden folglich nicht mehr von der Schleife erfasst). Durch die Abstraktion sieht man sofort, dass die innere Schleife überflüssig ist.

```

1  real_array_sweep2( chunk : ARRAY_CHUNK ) is
2      local
3          head : ARRAY_HEADER
4      do
5          from head := chunk.first_header
6          until head = chunk.after_last_header
7          loop
8              if head.is_marked
9              then head.unmark
10             else
11                 collect_array_segment(chunk, head)
12                 recycle_array_segment(chunk, head)
13             end
14             head := head.next_header
15         end -- loop over segments
16         if cond2
17         then stuff3
18         else stuff2
19         end
20     end -- real_array_sweep2

```

Der aufmerksame Leser hat bemerkt, dass in der alten Version immer `stuff2` vor dem `return` ausgeführt wird, aber in der neuen Version wird die Schleife normal verlassen und folglich `if cond2 then stuff3 else stuff2 end` ausgeführt. Trotzdem sind die beiden Versionen äquivalent, denn `cond2` bedeutet “alle Segmente wurden freigesetzt”, aber vor dem `return` hatten wir ja gerade ein markiertes Segment angetroffen, also können nicht alle freigesetzt worden sein, und es wird korrekterweise der `else`-Part `stuff2` ausgeführt. Wir haben hier also nicht nur einen “Sprung” beseitigt, sondern auch Redundanz und Komplexität: `stuff2` kommt nur noch einmal vor und ist mit einer klaren Bedingung versehen.

Die dritte Schleife in der Originalroutine kann man bequem in eine eigene Routine auslagern:

```

1  collect_array_segment(chunk : ARRAY_CHUNK
2                          head : ARRAY_HEADER) is
3      require not head.is_marked
4      local h : ARRAY_HEADER

```

```

5      do
6          from
7              head.mark_free
8              h := head.next_header
9          until
10             h = chunk.after_last_header
11             or else h.is_marked
12         loop
13             head.size |= head.size + h.size
14             h.mark_free
15             h := h.next_header
16         end
17     ensure
18         -- Segment 'head' is concatenation
19         -- of all (old) following unmarked segments.
20     end -- collect_array_segment

```

An den restlichen Hilfsroutinen kann man nicht viel verbessern, außer durch Einführung kleiner Hilfsroutinen noch ein bisschen Redundanz beseitigen und alle Typzwänge (*type casts*) ausfaktorisieren. Diesen C-Code erspare ich aber dem Leser und zeige stattdessen die nach C zurückübersetzte Version obiger drei Routinen:

```

1  static void
2  array_sweep(array_chunk_t* chunk)
3  {
4      chunk->free_list_of_large := NULL;
5      if (cond1)
6          stuff1;
7      else
8          real_array_sweep(chunk);
9  } // array_sweep
10
11 static void
12 real_array_sweep(array_chunk_t* chunk)
13 /* REQUIRE exactly the used segments of 'chunk' are ARRAY_MARKED
14    no segment is in the free list
15    ENSURE marked segments are unmarked
16    (previously) unmarked segments are coalesced and put into the freelist
17    (via 'recycle_array_segment')
18    if all the chunk is free, it is put into the chunk freelist
19 */
20 {
21     array_header_t* head := array_chunk__first_header(chunk);
22     while (head < array_chunk__after_last_header(chunk))
23         // inv: head <= last
24         {
25             if (array_header__is_marked(head))
26                 array_header__unmark(head);
27             else {
28                 collect_array_segment(chunk, head);
29                 recycle_array_segment(chunk, head);
30             }
31             head := array_header__next_header(head);

```

```
32     };
33
34     if (cond2(chunk))
35         stuff3(chunk);
36     else
37         array_chunk__append_to_env(chunk);
38 } // real_array_sweep
39
40 static void
41 collect_array_segment(array_chunk_t* chunk,
42                      array_header_t* head)
43 {
44     require( ! array_header__marked(head) );
45
46     array_header__mark_free(head);
47     array_header_t* h := array_header__next_header(head);
48     while ( h == array_chunk__after_last_header(chunk)
49           || array_header__is_marked(head) )
50     {
51         array_header__set_size(head, array_header__size(h));
52         array_header__mark_free(h);
53         h := array_header__next_header(h);
54     };
55 };
```

Die vielen kleinen Routinen, die von dieser neuen Version aufgerufen werden, ergeben sich alle, indem man den entsprechenden Code aus der alten Version herausschneidet. Wie man sehen kann beschreibt der resultierende Code den Algorithmus auf einem sehr abstrakten Niveau: keine Typzwänge mehr, keine Zeiger-Arithmetik; fast wie in einer objektorientierten Sprache. Wie man auch sehen kann, wird dieser Grad an Abstraktion aber nur durch ein hohes Maß an Disziplin erreicht: es ist schon recht aufwändig all diese Kapselungsroutinen zu schreiben, und ihre langen Namen sind auch recht abschreckend. Das ist eben der Unterschied zwischen einer Programmiersprache, die Abstraktion *erlaubt*, und einer Sprache, die Abstraktion *unterstützt*.

Alles in allem hat dieses Beispiel wieder einmal gezeigt, dass man mit den einfachsten Mitteln der Strukturierten Programmierung zusammen mit großzügig eingesetzten Routinen schon viel erreichen kann. Einfachheit eben.

6.4 Abstrakte Datentypen

6.4.1 Module

Ein Modul kann man sich in Analogie zu einer Routine wieder als einen Bereich mit lokalen Namen vorstellen, nur dass diesmal die lokalen Konstrukte selbst Routinen sind. In imperativen Sprachen kommt zu den lokalen Routinen auch noch ein lokaler Zustand hinzu (also Variablen, die pro Modul verfügbar sind), aber dazu kommen wir im nächsten Abschnitt.

Als Schnittstelle eines Moduls gibt man einfach eine Untermenge ihrer Routinen an. Hierbei stellt sich natürlich gleich die Frage, nach welchem Kriterium ein Modul seine Routinen exportieren oder geheim halten sollte. Als ich als Programmierlehrling erstmals mit Modulen in Berührung kam, stellte ich fest, dass ein Modul immer ein paar Hauptroutinen zu haben schien, die man in anderen Modulen braucht und wegen derer man das Modul aufruft, und dann gab es noch die Hilfsroutinen, die man nur programmiert, um sie von den Hauptroutinen aus aufzurufen, aber nicht vom restlichen Programm. Folglich exportierte ich damals stets die Hauptroutinen, und die Hilfsroutinen blieben geheim. Der Vorteil von Modulen bestand also

damals für mich nur darin, dass erstens “logisch zusammengehörige” Routinen im Programmtext nah beieinander standen, und das zweitens Hilfsroutinen verschiedener Module den gleichen Namen haben konnten, ohne dass es zu Konflikten kam (eben genau wie bei Routinen-lokalen Variablen).

Wenn man aber jetzt bedenkt, dass objektorientierte Sprachen ja qualifizierte Namensräume haben, so dass Namensüberschneidungen ausgeschlossen sind, welchen Grund sollte ich dann noch haben, eine Routine zu verstecken? Wenn jede Routine eine ordentliche Spezifikation hat (und sich auch danach verhält), dann kann doch der Kunde einer Klasse selbst entscheiden, welche Routine er aufrufen möchte! Damit der Kunde nicht von den Hilfsroutinen verwirrt wird, kann ich sie ja ans Ende der Schnittstelle stellen, so dass man die Hauptroutinen zuerst betrachtet. Aber warum sollte man Hilfsroutinen überhaupt verstecken?

Wie wäre es damit:

Erster Grundsatz der Informationsversteckung Module können einzelne Routinen verstecken (nicht exportieren), damit dadurch die Spezifikation der anderen Routinen einfacher wird.

Ich nehme an, dass jetzt mindestens 99% meiner Leser von diesem Postulat überrascht sind. Die Erklärung ist jedoch ganz einfach:

Wir stellen uns eine Abstrakte Datenstruktur vor (eine Menge, Sequenz, Abbildung), die durch einen Baum implementiert wird. Betrachten wir eine Hauptroutine, die ein Element in die Datenstruktur einfügt, und eine Hilfsroutine, die das Element im Wurzelknoten des Baumes zurück gibt. Wenn wir beide Routinen exportieren, müssen wir natürlich schon spezifizieren, wie sich ein Aufruf der einen Routine auf das Ergebnis der anderen Routine auswirkt. Man kommt hier nicht darum herum zu beschreiben, dass da etwas mit Bäumen passiert.

Wenn aber die Hilfsroutine versteckt ist, können wir die andere Routine so beschreiben, als wäre der Modulzustand wirklich eine Menge (oder was auch immer). Der einzige Effekt des Einfügens ist dann, dass das Element danach drin ist (was man mit einer speziellen Anfrage beobachten kann).

Dasselbe Spiel funktioniert mit Datenstruktur-Implementierungen, die eine Anfrage `kapazitaet` haben (und möglicherweise auch noch `setze_kapazitaet`). Dann muss die Spezifikation des Einfügens (und auch den Löschens usw.) natürlich sagen, was passiert, wenn die Struktur voll ist (`groesse = kapazitaet`). (In schlechten Bibliotheken werden solche Details oft weggelassen, aber dann kann man natürlich diese Routinen nicht mehr verlässlich einsetzen.) Ohne die Kapazität sind die Spezifikationen einfacher, und damit ist die Klasse einfacher zu verwenden.

Allgemein kann man also sagen, dass die Menge aller Anfragen einer Klasse einen Wertezustandsraum beschreiben (in dem je nach vorhandenen Befehlen vielleicht nicht alle Zustände eingenommen werden können). Je kleiner dieser Zustandsraum ist, desto einfacher ist die Klasse zu verwenden, denn erstens muss ja jede Spezifikation eines Befehls den kompletten Nachzustand beschreiben, andererseits muss sich der Benutzer einer Klasse immer über deren aktuellen Zustand im klaren sein — denn sonst kann er ja nicht sicher sein, dass ihm zukünftige Anfragen auch das liefern, was er erwartet.

Logisch äquivalent könnten wir auch den Hilfsroutinen leere Spezifikationen geben, also Vorbedingung `False` (darf man nie aufrufen) oder Nachbedingung `True` (sie garantiert für gar nichts). Dann könnten wir auch die Hauptroutinen genauso einfach spezifizieren und nur aufeinander Bezug nehmen lassen. Dieses Verfahren ist aber quasi äquivalent zum Verstecken der Hilfsroutinen, denn kein Programmierer würde so eine Funktion je aufrufen. (Technisch ist diese äquivalente Darstellung aber trotzdem interessant, weil ein Programm zum statischen Prüfen von Verträgen, solche Aufrufe als Fehler finden würde. Man bräuchte dann gar kein Spezielles Sprachkonstrukt zum Verstecken von Dingen mehr.)

Man kann das Ganze auch funktional-algebraisch erklären: Die eigentliche Abstraktion wird erreicht, weil unterschiedliche Repräsentationen eines Wertes für den Anwender nicht mehr unterscheidbar sind, und dadurch gelten mehr allgemeine Gesetze — eine neue Algebra entsteht.

Beispiel: Wenn wir Mengen als Bäume implementieren, so führen die Ausdrücke $a \cap b$ und $b \cap a$ möglicherweise zu unterschiedlichen Bäumen, aber trotzdem natürlich zu den gleichen Mengen. Das Kommutativ-Gesetz gilt für Bäume eben nicht, aber für die Algebra der Mengen ist es ein Grundbaustein.

Ein anderer, wesentlicher Vorteil der Abstraktion entsteht, weil die Funktionen der Implementierung Invarianten aufrecht erhalten, die der Anwender nicht beachten muss, weil er sie gar nicht verletzen kann! Wenn nämlich der Anwender nur ganz bestimmte Funktionen aufrufen kann, dann können diese bei jedem Aufruf voraussetzen, dass beim vorigen Aufruf auch eine von ihnen gerufen wurde. (Und ganz zuerst eine Initialisierungsroutine, die auch zur Clique gehört.) Und wenn jede dieser Routinen (und die Initialisierung) die Invariante per Nachbedingung impliziert, so kann sie sie auch automatisch bei jedem Aufruf als Vorbedingung voraussetzen, ohne dass der Aufrufer die Vorbedingung absichern muss. Und genau dadurch wird die Spezifikation des Moduls (d.h. alles, was der Aufrufer wissen muss und will) signifikant einfacher. Wir halten fest:

Erster Grundsatz der Informationsversteckung Ohne Invarianten ist das Verstecken (Nicht-Exportieren) von Routinen aus Klassen sinnlos, denn es führt nicht zu einer signifikanten Vereinfachung eines Programms.

Bei meiner langen Beschäftigung mit vielen verschiedenen Programmiersprachen habe ich auch sehr viele Möglichkeiten kennen gelernt, Elemente von Klassen oder anderen Modulen zu verstecken, und oft habe ich Diskussionen gelesen, in denen man versuchte diese Mechanismen sicher gegen Umgehung zu machen, so als ob der Kunden-Programmierer ein böser Feind wäre, der auf gar keinen Fall die versteckten Routinen aufrufen darf. Aber nie wurde diskutiert, wozu dieses ganze Verstecken eigentlich gut sein soll! Haben diese Menschen irgendwo gelesen, dass *information hiding* die einzig wahre Modularisierungstechnik sei, und jetzt wollen sie so viel wie möglich verstecken? Der Erfinder der `public protected private` Hierarchie, die in C++ und JAVA verwendet wird, hat sich scheinbar gedacht “Wenn das Verstecken so eine gute Sache ist, dann sollte man auch Dinge vor seinen Erben verstecken können.” Aber wozu es wirklich gut ist, hat er sich wohl nie gefragt! Ist es nicht endlich einmal an der Zeit, all diese Bemühungen als völlig kontra-produktiv zu entlarven? Schließlich sind Programmierer keine kleinen Kinder, vor denen man gefährliche Werkzeuge verstecken muss.

Und nun können wir sagen, wodurch man Abstraktion in Programmiersprachen erhält:

Informationen, die ein Programmierer braucht, um ein Programm zu verstehen, drückt man in Form von Zusicherungen aus. Ob man diese nun explizit aufschreibt oder nicht, sie sind immer da.
Informationen verstecken im Sinne des *information hiding* besteht also darin, Zusicherungen zu verstecken.

Wir haben gesehen, wie man in Routinen Abstraktion erreicht, indem man die Informationen über den Zustand der lokalen Variablen versteckt. Auf dieselbe Weise hilft uns die Invariante in einem Modul Informationen über den Zustand der lokalen Variablen von einer exportierten Routine zur nächsten zu tragen, und ihn dadurch vor dem Aufrufer zu verstecken.

Sehr oft kommt es vor, dass ein Modul bestimmte Datentypen an seine Kunden zum Bearbeiten weiter gibt. Um Abstraktion zu erreichen, sollte das Modul wiederum Invarianten über diesen Datentyp durchsetzen. Dazu müsste man sicherstellen, dass jeder Zugriff auf so einen Datentyp

nur durch exportierte Routinen des Moduls selbst geschieht: der Kunde darf zwar den Datentyp “in die Hand nehmen” und von einer Routine zur nächsten geben oder zwischenspeichern, aber er soll nicht auf dessen Repräsentation zugreifen — diese wollen wir genauso schützen wie die lokalen Variablen eines Moduls. In pre-OO Sprachen löst man das Problem, indem ein Modul den Namen eines Datentypen exportiert, aber nicht dessen Struktur. In HASKELL geht das noch genau so.

Eine leichte Inkonvenienz entsteht dann dadurch, dass ein Modul für alle Instanzen eines oder mehrerer Datentypen verantwortlich ist: man muss den Datentyp immer als Parameter übergeben, man muss umständlich hineinschauen, und die eigentlichen lokalen Variablen des Moduls braucht man überhaupt nicht mehr. Die Lösung hierauf ist ganz einfach, das Modul und den Datentypen zusammenzulegen, um etwas neues, größeres entstehen zu lassen um ein flexibles, starkes Schema zu fassen ich rede von Klassen!

Eine *Klasse* ist ein Modul, das genau einen abstrakten Datentyp definiert.

Die lokalen Variablen dieses Moduls namens Klasse existieren genau einmal pro Instanz des Datentyps namens Klasse, und weil das schon etwas Besonderes ist, nennen wir sie ab sofort *Attribute* der Klasse. Willkommen in der Welt der Objektorientierung! Der Vorteil des Konzeptes: man programmiert das Modul einmal, so als gäbe es nur eine Instanz, aber dann kann man beliebig viele Instanzen davon erstellen. Das sind dann die kleinen Objekte.

Bei so einem genialen Konzept fragt man sich natürlich, wo es so plötzlich herkommt. Soweit ich weiß, hat tatsächlich schon die Sprache SIMULA 67 alle wichtigen Zutaten besessen (insbesondere Vererbung, aber soweit sind wir hier jetzt noch nicht), aber natürlich hat man nicht sofort die ganze Größe der Idee erkannt. Dazu brauchte es erst einmal noch einer spielerischen Phase mit der Sprache SMALLTALK, ganz ohne an Konzepte wie Module oder ADTs, ja nicht einmal Routinen zu denken, die hießen dort “Methoden”, und der Begriff “Objekt” stammt auch aus dieser spielerischen Phase. SMALLTALK war als Sprache für Kinder konzipiert. Aus SMALLTALK ging auch die Bewegung des Extremen Programmierens hervor. (Wahrscheinlich, weil es so viel Spaß gemacht hat, SMALLTALK verfügte nämlich für die damalige Zeit auch schon über hervorragende graphische Entwicklerwerkzeuge.)

SMALLTALK hat mit seiner kindlichen Manier einigen Computerfreaks den Kopf verdreht, so dass in den 80ern mehrere neue OO Sprachen entwickelt wurden. Eine davon basierte auf C und hat mit ihrem Erfolg bei Programmierern Tausende Projekte scheitern lassen, Millionen fehlerhafter Programme hervorgebracht und die Industrie einige Milliarden an Verlusten gekostet — kurz und gut, sie hat die Softwaretechnik um Jahrzehnte zurückgeworfen, aber das machte damals nichts, denn Computeranwendungen erlebten einen unheimlichen Aufstieg und selbst mit schlechten Programmen machte man noch Gewinne [50]. (Man musste also gar keine guten Programme schreiben und brauchte folglich auch keine gute Programmiersprache.)

Anno 1985 hat es dann die Sprache EIFFEL geschafft, die Flexibilität von SMALLTALK mit einer strukturierten Methode zu verbinden. Soweit ich weiß, war es auch die erste Sprache mit Abstrakten Datentypen, aber auf jeden Fall die erste, die ADTs konsequent mit Modulen zu Klassen verbindet. Obiger Merksatz stammt sinngemäß von Bertrand Meyer [29]. Es war sowieso eine verwunderliche Zeit, in der es so viele schlechte Bücher über OO und so viele schlechte Sprachen gab, aber Meyer produziert das beste Buch und die beste Sprache — und das sind sie sogar heute noch, weil der Computerboom in den Internetboom überging, welcher gleichsam gute Softwaretechnik unnötig machte. Und so bleibt EIFFEL die einzige Programmiersprache mit Verträgen, die einzige ernsthafte Programmiersprache.

Als kleines Intermezzo vor dem nächsten Abschnitt hier noch eine andere tolle Anwendung von Klasseninvarianten, um Zusicherungen bequemer durch Programme zu tragen. Man stelle sich vor, dass verschiedene Routinen die Aussage $i \geq 0$ für einen Routinen-Parameter i in der Vorbedingung enthalten. (Der Trick funktioniert

auch mit anderen Aussagen, solange sie sich nur auf einen Parameter beziehen.) Ein fortschrittlicher Anwender von OO-Sprachen kommt natürlich sofort auf die Idee, diese Vorbedingung doch lieber mit dem Typsystem auszudrücken. Wenn i vom Typ `INTEGER` ist, kann man eine neue Klasse `NONNEGATIVE_INTEGER` einführen, oder etwas positiver formuliert `NATURAL`.

Was die meisten OO-Anwender aber schon nicht mehr wissen ist, dass sich die Klasse `NATURAL` genau durch ihre Invariante `Current >= 0` von `INTEGER` unterscheidet! Und wer diese Nutzung des Typsystems als allgemein einsetzbares Muster zur Umwandlung von Zusicherungen in statische Prüfungen sieht, irrt leider auch. Denn natürlich muss die Klasse `NATURAL` bei jeder Objekterstellung ihre Invariante sicherstellen, was in diesem Fall heißt, dass die Erstellungsroutine ein Argument $i : \text{INTEGER}$ hat und die Vorbedingung $i >= i$. Da haben wir wieder die Laufzeit-Prüfung, und sie wird bei jeder einzelnen Erstellung eines `NATURAL`-Objektes geprüft! Wir sehen also, dass sich dieser Trick nur lohnt, wenn wir mit den Objekten nicht viel rechnen, sondern sie nur weiterreichen.

6.4.2 Prozeduren versus Funktionen

Wir haben im Abschnitt 6.1 Funktionen und im Abschnitt 6.2 Prozeduren kennen gelernt; im vorigen Abschnitt haben wir von beiden als Routinen gesprochen und Attribute hinzugefügt. Aus Sicht des Kunden einer Klasse ist es nützlich, alle *Elemente* einer Klasse in Anfragen und Befehle aufzuteilen. Abbildung 6.1 aus [29] stellt den Zusammenhang der Begriffe dar. Bemerkenswert nützlich ist, dass ein Kunde bei den Anfragen ein Attribut nicht von einer Funktion (ohne Parameter) unterscheiden kann. Klassen exportieren Attribute genauso wie Funktionen nur zur Anfrage. Verändern kann man Attribute nur indirekt über Befehle (Prozeduren), so dass es einer Klasse immer möglich ist, ihre Invariante zu verändern, ohne dass Kundenklassen geändert werden müssen. (Dieses Argument findet man sonst auch nicht in der Literatur!).

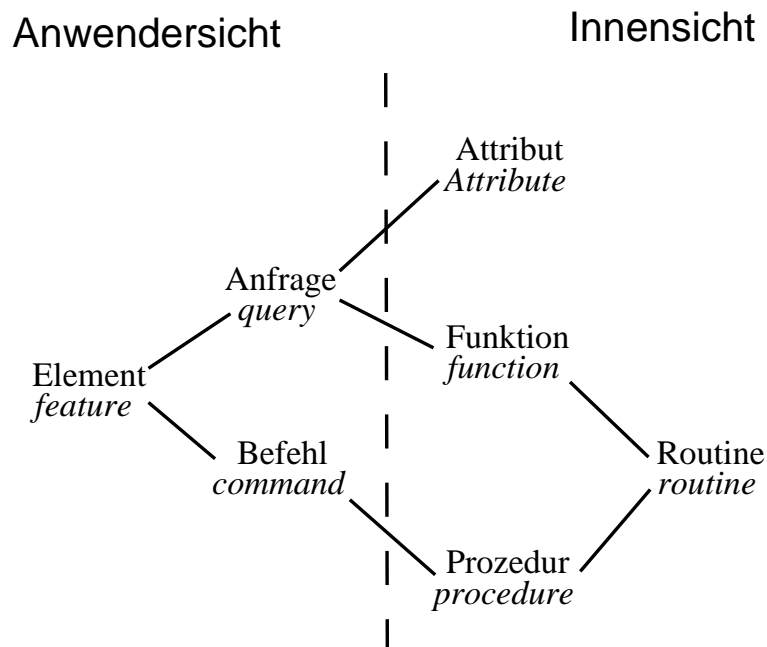


Abbildung 6.1: Terminologie der Schnittstellenelemente

Das Grundprinzip, dem wir uns in diesem Abschnitt widmen, ist das Fundament für die Vereinigung von funktionalem und objektorientiertem Programmieren.

Trennung von Anfrage und Befehl Befehle ändern den abstrakten Zustand von Objekten und haben keinen Rückgabewert. Anfragen haben einen Rückgabewert und ändern den abstrakten Zustand von Objekten nicht. Jedes Element einer Klasse ist entweder Anfrage oder Befehl.

Da Prozeduren sowieso keine Werte zurückgeben können, und jede Routine entweder Funktion oder Prozedur ist, geht es eigentlich nur darum, dass Funktionen den abstrakten Zustand des Zielobjekts (und auch keines anderen) nicht ändern dürfen. Kurz: Funktionen dürfen keine sichtbaren Seiteneffekte haben. Oder in den Worten von Bertrand Meyer: “Eine Frage zu stellen, soll die Antwort nicht verändern.”

Dieses Prinzip erlaubt es uns, Funktionen genauso frei zu verwenden wie in der funktionalen Programmierung, obwohl wir die Funktionskörper durchaus auch imperativ implementieren können. Den Begriff “abstrakter Zustand” eines Objekts hatten wir noch nicht explizit erwähnt, aber es geht dabei nur um die Informationen, die man von allen möglichen Anfragen des Objekts bekommen kann. Wenn der abstrakte Zustand nicht geändert werden soll, heißt das ganz pragmatisch, dass alle diese Anfragen (für alle möglichen Argumentwerte) ihr Ergebnis nicht verändern dürfen. Normalerweise hat man aber zu einer Klasse eine Art Modell im Kopf, das diesen Anfragen vorgibt, was sie tun sollen. Im besten Fall repräsentiert der Objekt-Zustand einen einfachen oder zusammengesetzten mathematischen Wert (wie zum Beispiel eine rationale Zahl, oder eine Menge, wieder einmal), und die Anfragen beziehen sich alle darauf. Dann heißt “frei von sichtbaren Seiteneffekten”, dass dieser Modellzustand nicht verändert wird. In anderen Fällen hat man zumindest Basisanfragen (wie Attribute) und daraus abgeleitete Anfragen, so dass man nur erstere nicht verändert, und dann verändern sich die anderen natürlich auch nicht. Für alle, die es noch nicht wissen: das Standardbeispiel für einen nicht sichtbaren Seiteneffekt ist eine Funktion, die ein Attribut als Zwischenspeicher *cache* verwendet, um zukünftige Anfragen schneller zu beantworten. Eine Wertänderung dieses Attributs ändert nicht die Ergebnisse zukünftiger Anfragen, es macht sie nur schneller. In dieselbe Kategorie fällt auch eine Reorganisation einer Datenstruktur (zum Beispiel Rebalancieren eines Baumes).

algebraische Klassen Wenn der Wert einer Anfrage einer Klasse von keinem Befehl der Klasse verändert wird, so sprechen wir von einer *unveränderlichen* Anfrage. Das Schlüsselwort `immutable` in der Deklaration einer Anfrage `x` fügt automatisch jedem Befehl der Klasse die Nachbedingung `x = old x` hinzu. (Diese Bedingung lässt sich leider nicht so leicht prüfen, wie es hier aussieht, denn falls die Anfrage Parameter hat, müssen diese allquantifiziert werden.) Unveränderliche Anfragen können trotzdem noch vom Zustand des Objekts abhängen, aber eben nur von einem Teil, der sich seit der Erstellung des Objektes nicht verändert hat. Ein häufiges Muster für unveränderliche Anfragen sind Attribute, denen nur einmal bei Erstellung des Objekts ein Wert zugewiesen wird.

Wenn eine Klasse nur aus Anfragen besteht, dann sind diese natürlich alle unveränderlich (weil kein Befehl da ist, um sie zu ändern). Man kennzeichnet dann nicht jede Anfrage mit `immutable`, sondern gleich die ganze Klasse. Die Objekte derart unveränderlicher Klassen repräsentieren keine Module mit Zustand mehr, sondern sie repräsentieren direkt einen Wert, so wie in der funktionalen Programmierung. Zahlen und Zeichenketten sind solche Werte, aber auch Datumswerte oder Adressen oder sogar unveränderliche Datenstrukturen wie Mengen oder Abbildungen.

Da man Objekte algebraischer Klassen nicht verändern kann, erstellt man häufig neue. Dies geschieht in der Regeln nicht direkt über einen Konstruktoraufruf, sondern indirekt, weil das Objekt Ergebnis einer gerufenen Anfrage ist. Klassisches Beispiel ist der Ausdruck `1 + 1`, der das neue Objekt `2` liefert. Oftmals kann man auch auf das Erstellen neuer Objekte verzichten, weil man stattdessen eine Referenz auf ein Objekt mit dem gleichen Wert liefern kann. Dieses Muster ist besonders schonend für den Speicherverbrauch.

Das Schlüsselwort `immutable` existiert in keiner offiziellen Version von EIFFEL. Ich hatte

es im letzten Jahr für einen Artikel erfunden, in dem ich die funktional-objektorientierte Programmierung beschrieb, so wie sie hier dargelegt ist. Der Artikel wurde aber nie fertig. Grund für die Erfindung waren übrigens Probleme mit dem Kopieren von Objekten im Zusammenhang mit EIFFELS `expanded` Schlüsselwort. Mit letzterem kann man Objekte direkt in Variablen speichern an Stelle einer Referenz auf das Objekt. Problematischerweise hat dieses Schlüsselwort aber sowohl Auswirkungen auf die Effizienz als auch auf die Semantik eines Programms, weil damit auch automatisch zwischen Referenz-Zuweisung und Kopier-Zuweisung umgeschaltet wird. Das Schlüsselwort `immutable` kann diese Probleme teilweise lösen, weil bei unveränderlichen Klassen beide Zuweisungssemantiken den gleichen Effekt haben — man kann dann also den Compiler die effizientere auswählen lassen!

6.4.3 Zum Beispiel Iteratoren

```
class JAVA_ITERATOR[T] is
export next, has_next
feature
  has_next : BOOL

  next : T is
    -- Advances iterator
    -- and returns current element.
  require
    has_next
end -- JAVA_ITERATOR
```

```
class ITERATOR[T] is
feature
  off : BOOL

  item : T is
    -- Current element.
  require
    not off

  forth is
    -- Advance iterator.
  require
    not off
  end
end -- ITERATOR
```

Abbildung 6.2: Welcher Iterator hat die einfachere Schnittstelle?

Zunächst einmal habe ich eine Weile gebraucht, um mich zu versichern, dass die beiden Iteratoren wirklich immer “funktionieren”, insbesondere auch bei leeren Folgen! Anschaulich kann man sich vorstellen, dass der JAVA-Iterator immer “zwischen zwei Elementen” steht und beim Wechsel der Position das Element zurück liefert, dass er dabei überspringt. Der EIFFEL-Iterator hingegen steht immer “auf einem Element”, und dieses Element kann man so oft abfragen, wie man möchte. Im Fall einer leeren Folge steht der JAVA-Iterator in der Lücke hinter dem letzten Element, und der EIFFEL-Iterator steht auf einem undefinierten Element hinter dem letzten Element. Dieses “undefinierte Element” mag zunächst wie eine Verkomplizierung erscheinen, aber wenn man genauer hinschaut, ist die Semantik der Schnittstelle weiterhin einfach. Letzten Endes kann man sich auch von der korrekten Spezifikation der Schnittstellen überzeugen, indem man sie ganz formal betrachtet: im Fall `not has_next` (für JAVA) bzw. `off` (für EIFFEL) sind die anderen Klassenelemente nicht definiert, es gibt also keine Elemente mehr ab der aktuellen Position. Im Falle einer leeren Folge, wird der Iterator diesen Zustand schon am Beginn haben, also funktioniert das auch.

Übrigens haben wir hier mal eine Stelle an der C++ dem JAVA überlegen ist: die dortigen Iteratoren haben auch zwei separate “Routinen” `it++` zum Weitersetzen und `*it` zum Anfragen des Wertes. Allerdings machen das C++-Programmierer oft zunichte, indem sie schreiben: `*it++` — und der Leser fragt sich jedes Mal, ob das nun den Wert vor oder nach dem Weitersetzen liefert (ich müsste es auch erst nachschlagen). Dieses Konstrukt demonstriert den Fehler, Routinen mit Seiteneffekt in Ausdrücken zu verwenden. Und dazu kommt noch, dass die Struktur des Ausdrucks nicht klar ist: `(*it)++` oder `*(it++)`? Wir haben hier also eine schlechte Semantik gepaart mit einer mehrdeutigen Syntax, die gemeinsam viel

Leid bei Programmierern verursachen können. Wer mehr davon haben möchte, sollte PERL programmieren.

Auf den zweiten Blick ist C++ aber mit einem anderen Konstrukt bis unten in der Hierarchie des Schreckens: seine *streams* vereinigen die drei Elemente *off*, *item*, *forth* in einem einzigen namens *>>*. In 95% Prozent alle Fälle erreicht man mit einem `while (cin>>i)` genau was man braucht (mit zwei Seiteneffekten *und* einem Rückgabewert), aber in den restlichen Fällen wird die Benutzung so schwierig, dass man sich über häufig abstürzende C++ Programme nicht wundern braucht. C++ glänzt hier mit seinem Motto: “Verwirre den Programmierer, indem Du zwei verschiedene Konstrukte (nämlich Iteratoren und *streams*) anbietest, die fast aber doch nicht ganz das selbe tun und keinerlei Hilfe bieten, wann denn nun welches anzuwenden sei.”

```
while (it.has_next)
{
    // call 'it.next'
    // exactly once
}
```

```
until it.off
loop
    -- use 'it.item'
    -- as often as you like
    it.forth
end
```

Abbildung 6.3: Welcher Iterator lässt sich leichter verwenden?

Mit den beiden Code-Mustern können wir die Pragmatik der Iterator-Schnittstellen vergleichen. Wir sehen, dass die EIFFEL-Variante offensichtlich eine Zeile mehr braucht, und dass dafür der Schleifenkörper so oft, wie er mag, auf den Wert des Iterators zugreifen kann. Insbesondere kann er es auch überhaupt nicht tun, falls er keine Lust hat — er muss also keinen zusätzlichen Spezialfall betrachten!

Der JAVA-Programmierer hingegen wird zunächst sagen: “wenn ich den Wert mehrmals benutzen will, packe ich ihn einfach in eine zusätzliche Variable; das kostet mich eine Zeile

```
Object val = it.next,
```

aber nur dann, wenn ich sie auch wirklich brauche.” Auf lange Sicht gewinnt aber in jedem Fall der EIFFEL-Programmierer: er kann nämlich den Schleifenkörper in jedem Fall *unabhängig* vom drumherum betrachten: `it.val` kann er wie eine ganz normale Objekt-Referenz benutzen (z.B. genau wie einen formalen Routinen-Parameter). Dieser Vorteil zeigt sich bereits, wenn in der Schleife eine Fall-Unterscheidung stattfindet: der JAVA-Programmierer muss in *jedem der beiden Fälle* sicherstellen, dass `it.next` genau einmal gerufen wird — der EIFFEL-Programmierer kümmert sich hingegen ganz ruhig um sein eigentliches Programmierziel.

Insgesamt hat der JAVA-Programmierer noch Glück, dass die Iterator-Angelegenheit nicht über Routine-Grenzen hinweg übertragen wird. Mit der Wertübergabe (*call-by-value*) von OO Sprachen ist es egal, wie oft der Iterator-Wert innerhalb der gerufenen Routine verwendet wird — der formale Parameter übernimmt die Rolle der Hilfsvariable. Insofern kann man sich in JAVA vor bösen Iterator-Überraschungen schützen, indem man nicht-triviale Schleifenrumpfe in eine Hilfsroutine packt — oder man verwendet in jedem Fall die Hilfsvariable (sozusagen als Bestandteil des Code-Musters). Das hat dann auch noch den Vorteil, dass man notwendige Type Casts nur an einer Stelle durchführt. (Da der Original JAVA-Iterator nämlich in JAVA programmiert ist, und nicht in EIFFEL, hat er natürlich keinen generischen Typ.) Das wahre Code-Muster in JAVA ist dann also:

```
while (it.has_next)
{
    SomeClass val = (SomeClass)it.next;
    // use 'val' as often as you like
}
```

Fassen wir das von den Iteratoren Gelernte zusammen: ein Entwurfs-Konflikt bestand zwischen dem Prinzip “Spezifiziere das Einfachste für Deine konkrete Anwendung” und der Heuristik

“Suche Einfachheit in Verallgemeinerungen”. Normalerweise hat das Prinzip Vorrang, weil man sich ja nicht alle möglichen Wiederverwendungen im Vorhinein überlegen kann. In unserem Fall wird der Konflikt aber von einer ganz konkreten Regel gelöst: der TRENUNG VON FRAGE UND BEFEHL. Diese Regel bevorzugt Einfachheit durch Allgemeinheit in ganz bestimmten Fällen, basierend auf langer Erfahrung, und nicht nur meiner.

Die *Heuristik* gibt uns einen Hinweis, in welche Richtung wir nachdenken sollten. Das *Prinzip* hilft uns, Alternativen zu vergleichen. Und die *Regel* sagt uns—in den ganz konkreten Fällen, wenn sie anwendbar ist— was wir ganz konkret tun (oder nicht tun) sollten.

6.4.4 Abstrakte und andere Datentypen

Unsere Methodik unterscheidet drei Arten von Datentypen:

direkte Datentypen Deren Funktionen werden direkt von der Hardware zur Verfügung gestellt. Man kann sie also benutzen, ohne sie zuvor zu definieren, und sie sind auch entsprechend effizient. Allerdings spiegeln sie nur einen sehr kleinen Teil der Anwendungswelt wider und dies auch noch mit willkürlichen Restriktionen (z.B. gibt es nur ein begrenztes Intervall von ganzen Zahlen).

zusammengesetzte Datentypen Programmiersprachen bieten verschiedene Möglichkeiten, die primitiven Datentypen zu komplexeren zu kombinieren, und damit kann man sich beliebig nah an die Anwendungswelt heran arbeiten. Die Sprachen bieten auch verschiedene Konstrukte, um die komplexen Typen als Summe ihrer Teile aufzufassen.

abstrakte Datentypen Hierfür stellt die Programmiersprache nur ein leeres Gerüst bereit, dass der Programmierer dabei nach Belieben ausfüllen kann. Weil man aber aus nichts auch nicht mehr machen kann, müssen abstrakte Datentypen mit Hilfe schon existierender Datentypen implementiert werden. Die Besonderheit besteht nun darin, dass die Implementierung der Datenstruktur genauso vor dem Anwender versteckt wird, wie die einer Prozedur oder Funktion.

Zusammengesetzte Datentypen wurden erstmals umfassend in Hoares “Notes on Data Structuring” vorgestellt. Auf dieser Darstellung basierte später Niklaus Wirths Programmiersprache PASCAL, die quasi einen Standard auf diesem Gebiet setzte. In funktionalen Programmiersprachen erlaubt es die algebraische Natur der Sprache die beiden wichtigsten Kompositionsmechanismen deutlicher herauszuarbeiten, und in Spezifikationssprachen wie Z und B kann man direkt die Wertemengen beschreiben, die von zusammengesetzten Typen bestimmt werden...

als Mengen	als algebraische Datentypen	in Pascal	objektorientiert
Kreuzprodukt	Produkt	Record	Klassenattribute
Vereinigung	Summe	Variant	dynamische Bindung

All den vorgenannten Sprachen liegt also ein konstruktivistischer Ansatz zu Grunde; die Abstraktion wurde oft erst nachträglich eingeführt und auch nur in Form von zusätzlicher Funktionalität zum Verstecken von Implementierungen. Im objektorientierten Ansatz hingegen sind die abstrakten Datentypen Standard (sogar die primitiven Datentypen werden als Klassen definiert), und die beiden Kompositionsmechanismen verlieren sich im Arsenal der objektorientierten Ausdrucksmittel.

Zahlen und Zeichen sind die wichtigsten direkten Datentypen, die in jeder Programmiersprache vorhanden sind. Zeichenketten werden meist als zusammengesetzte Datentypen dargestellt: unter Verwendung von Feldern oder Listen. In modernen Sprachen wie EIFFEL sind Zeichenketten abstrakte Datentypen, man kann aber noch einen

Schritt weiter gehen und sie durch Vererbung gleichzeitig als zweierlei auffassen: als abstrakten Datentyp oder als Zusammensetzung eines direkten Typs (Zeichen) mit einem abstrakten Typ (Listen oder Sequenzen). Und da sind wir auch schon beim nächsten Punkt: man kann Zeichenketten entweder als algebraische Datentypen auffassen oder als abstrakte Maschinen, die jeweils eine Zeichenkette zur Manipulation bereithalten. Rein funktionale Sprachen kennen natürlich nur ersteres, EIFFEL kennt leider nur letzteres, aber JAVA (und das ist ihr einziger Vorsprung) kennt beides und das sogar unter guten selbst-beschreibenden Namen: `STRING` und `STRINGBUFFER`.

Ein anderes Beispiel für direkte Datentypen sind verkettete Listen und Bäume, wie sie besonders in der funktionalen Programmierung gern verwendet werden. Darauf aufbauend kann man abstrakte Mengen, Sequenzen und Abbildungen implementieren — dies sind dann abstrakte Datentypen. Dies ist allerdings schon ein Schritt, den viele Programmierer nicht gehen: sie verwenden lieber direkt Listen und Felder, für die ja bereits schon *so viele* Funktionen zur Verfügung stehen. Die Anwendungsprogrammierer fügen dann selbst noch Funktionen hinzu, zum Beispiel zum Einfügen von Elementen, falls sie nicht schon drin sind, oder zum Verbinden zweier Felder oder Listen, wobei gleiche Elemente nur einmal behalten werden. Beide Operationen passen aber eigentlich besser zu einem Mengentyp, und indem der Programmierer die Mengen hier quasi nebenbei implementiert, erzeugt er nicht nur ein Durcheinander von Funktionen, die alle auf Listen/Feldern arbeiten, sondern er geht auch das Risiko ein, einmal versehentlich seine implizite Invariante zu verletzen (nämlich, dass jedes Element nur einmal vorkommt). Aber wenigstens stehen ja heutzutage die wichtigsten abstrakten Datenstrukturen schon als Bibliotheken zur Verfügung, so dass Programmierer nur noch die richtige Auswahl treffen müssen.

Wenn man zusammengesetzte Datentypen als Synthese sieht, so sind abstrakte Datentypen die Analyse: erstere kombinieren existierende Datentypen mit all ihren Eigenschaften, letztere beginnen bei Null und fügen nur hinzu, was explizit gewünscht wird. Der Unterschied zwischen zusammengesetzten und abstrakten Datentypen ist elementar für den Programm-Entwurf: zusammengesetzte Typen erlauben zunächst eine große Wiederverwendung, nämlich ihrer Bestandteile, mit denen man leicht sehr komplexe Strukturen bauen kann — aber auch *zu* komplexe Strukturen. Entweder enthalten die Strukturen zu viele technische Aspekte, die man vor möglichst viel Code verstecken sollte, oder sie vermischen verschiedene fachliche Sichten, die man lieber trennen sollte.

Die Trennung von Aspekten findet durch Schnittstellen statt. Im Idealfall definiert jede Schnittstelle einen Abstrakten Datentyp, den man mit einer Klasse implementieren kann. Wenn zu der Schnittstelle eine mathematische oder technische Abstraktion passt (zum Beispiel eine abstrakte Datenstruktur), liegt dieser Idealfall vor. Aber gerade beim Formalisieren von komplexen Anforderungen sind Funktionalitäten oft ADT-übergreifend. Es kommt dann auch vor, dass Klassen keine neue Abstraktionsschicht einführen, sondern mit ihren Attributen und `set_xxx`-Prozeduren ähneln sie eher einem zusammengesetzten Datentyp (nämlich einem Record). Genau wie wir es schon von Routinen gesehen haben, werden die Klassen hier wie Definitionen verwendet. Erst wenn man später gewisse Dinge hinzufügt oder optimiert, finden sich Informationen zum Verstecken, und die Schnittstelle bringt wieder eine echte Abstraktion.

Die zusammengesetzten Datentypen spielen wiederum bei der Konstruktion der Schnittstellen eine Rolle: man hat die Wahl, einfache zusammengesetzte Datentypen als Parameter zu verwenden oder die notwendige Funktionalität in speziellen Routinen zu implementieren. So kann zum Beispiel eine Anfrage eine Liste von Elementen zurückgeben, aber alternativ kann man direkt Funktionen zum Traversieren der gespeicherten Elemente anbieten.

6.5 Fallstudie: flexible Arrays

In modernen objektorientierten Programmiersprachen bietet es sich an, Arrays als abstrakte Datentypen zu implementieren. Im einfachsten Fall hat dieser Typ eine Konstruktionsroutine mit einem Parameter für die gewünschte Größe des Arrays. Diese Art von dynamischen Arrays stellt eine einfache Schnittstelle zur Speicherverwaltung des Laufzeitsystems her und Compiler können auch statische Implementierungen bieten, falls eine Array-Variable offensichtlich nur einmal allokiert wird. Hier ist die minimale Schnittstelle solcher Arrays:

```
class NATIVE_ARRAY[T] creation make feature
  size : INTEGER

  make( nsize : INTEGER )
    ensure size = nsize

  put( i : INTEGER; x : T )
    require 0 <= i and i < size
    ensure at(i) = x

  at( i : INTEGER ) : T
    require 0 <= i and i < size
```

Auf der anderen Seite kann man diese Schnittstelle verwenden, um verschiedenste andere Datenstrukturen zu implementieren (Mengen, Tabellen, Listen, ...). Eine Zwischenstufe auf diesem Weg sind oft flexible Arrays: diese können zusätzlich ihre Größe in beliebigen Schritten verändern. Man kann flexible Arrays mit Hilfe der einfachen dynamischen Arrays implementieren; diese Implementierung ist auf einem höheren Abstraktionsniveau und kann unabhängig vom Laufzeitsystem (und dessen Optimierungen) geschehen.

Ein flexibles Array verwendet klassischerweise zur Implementierung ein dynamisches Array, dass immer etwas zu groß gewählt wird, damit die meisten Größenänderungen keine Neuallokierung erfordern. Wenn man dann neu allokiert, so will man das zu Grunde liegende Array mindestens verdoppeln oder halbieren, damit sich das Kopieren der Elemente lohnt. Wir kommen dadurch zu folgender Invariante:

```
size = 0 implies storage = Void
and size > 0 implies ( size <= storage.size and storage.size < 4*size )
```

Die Fallunterscheidung dient dabei nur der Optimierung für den Fall, dass $size = 0$. Der wesentliche Teil besteht aber in der Ungleichung, die die Größe des zu Grunde liegenden Arrays beschränkt.

Man mag sich fragen, ob der Aufwand für die Optimierung des Falls $size = 0$ gerechtfertigt ist. Dabei muss man bedenken, dass auch die Allokierung eines *leeren* dynamischen Arrays seitens des Laufzeitsystems Speicher verbraucht: ob mit Garbage Collection oder ohne. Falls also irgendwann mal jemand eine ganze Menge leerer flexibler Arrays erstellt, um dann nur einige davon zu füllen, so kann sich allein dieser Verwaltungsspeicher zu einem kritischen Faktor entwickeln.

Wenn man eine Routine zum Ändern der Größe des flexiblen Arrays schreibt, so führt die Optimierung zu vier unterscheidenden Fällen: jeweils vor und nach der Größenänderung kann die Größe Null sein. Durch die explizite Invariante kann man den Code aber vereinfacht schreiben und trotzdem leicht verifizieren:

```
set_size( s : INTEGER ) is
  local
```

```

new_storage : BOB_NATIVE_ARRAY[T]
i : INTEGER
do
  if s = 0 then storage := Void
  elseif
    storage = Void or else
      (s > storage.size or storage.size >= 4*s)
  then
    create new_storage.make(s*2)
    from i := 1 until i > size.min(s)
    loop
      new_storage.put(i - 1, at(i))
      i := i + 1
    end
    storage := new_storage
  end
size := s
end -- set_size

```

Wenn die Größe auf Null gesetzt werden soll, ist egal ob vorher ein Array allokiert war oder nicht; im ersten Fall ist die Löschung schlicht ohne Effekt. Wenn vorher und nachher die Größe ungleich Null ist, müssen wir nur dann aktiv werden, wenn die neue Größe nicht zum bereits reservierten Speicher passt: neuen Speicher allokiert und die Elemente dorthin kopieren. Falls vorher noch gar kein Speicher reserviert war, müssen wir genau dasselbe tun, also fallen die Fälle zusammen, nur müssen wir aufpassen, dass auf `storage` nicht zugegriffen wird, wenn kein Objekt allokiert ist; die Verwendung von `or else` und die Schleifenbedingung stellen genau das sicher.

Man kann sich jetzt leicht überlegen, dass diese klassische Implementierung maximal den vierfachen Speicher belegt, und dass man ohne übermäßige Reallokierung nur dann weniger Verschwendung erreichen kann, wenn man den zu Grunde liegenden Speicher in mehreren Teilen allokiert. Trotz der Einfachheit dieser neuen Idee und der Bedeutung flexibler Arrays wurde eine komplette Analyse dieses Weges erst 1999 vorgestellt [3]. In dieser Lösung haben wir eine ganze Liste zu Grunde liegender Arrays, genannt `blocks`, alle von der Größe `block_size`. Die Invariante dieser Datenstruktur lautet wie folgt:

```

size.max(1) <= block_size^2 and block_size^2 < (size * 16).max(2)
and blocks.size - 1 = (size - 1) div block_size -- (*)

```

Die erste Zeile drückt ganz einfach aus, dass `block_size`, die Größe der Blöcke, ungefähr der Wurzel der Array-Größe entspricht. Folglich ist auch die Anzahl der Blöcke in dieser Größenordnung. In [3] findet man den (leichten) Beweis, dass dadurch die Platzverschwendung minimal ist. Übrigens sorgt die Verwendung von `max` dafür, dass die Bedingung auch bei Größen von 0 und 1 erfüllbar ist, so dass wir einen Sonderfall sparen.

Die letzte Zeile (*) bestimmt `blocks.size`, die Anzahl der allokierten Blöcke; als Erklärung kann man sich überlegen: wenn der letzte Block genau ein Element hat, gilt

$$size = block_size \times (blocks.size - 1) + 1$$

und falls alle Blöcke voll sind

$$size = block_size \times blocks.size$$

(Diese beiden Aussagen kann man auch als Teil der Invariante formulieren und automatisch prüfen lassen.)

Wir haben also für alle Zustände

$$\mathit{block_size} \times (\mathit{blocks.size} - 1) + 1 \leq \mathit{size} \leq \mathit{block_size} * \mathit{blocks.size}$$

und das ist äquivalent zu:

$$\mathit{size}/\mathit{block_size} \leq \mathit{blocks.size} \leq (\mathit{size} - 1)/\mathit{block_size} + 1$$

und wiederum zu:

$$\mathit{size}/\mathit{block_size} - 1 \leq \mathit{blocks.size} - 1 \leq (\mathit{size} - 1)/\mathit{block_size}$$

Diese letzte Aussage ist zu (*) äquivalent, wegen der einfachen Regeln

$$q = a \div b \equiv a/b - 1 < q \leq a/b$$

und

$$(\mathit{size} - 1)/\mathit{block_size} < \mathit{size}/\mathit{block_size}$$

Man kann sich die Bedingung (*) aber auch direkt überlegen (wir nummerieren hier Elemente und Blöcke ganz informatisch ab 0): Das *i*te Element des flexiblen Arrays wird stets gespeichert im Block $i \div \mathit{block_size}$. Das letzte Element $\mathit{size} - 1$ also im Block $(\mathit{size} - 1)/\mathit{block_size}$.

Folglich haben wir $(\mathit{size} - 1)/\mathit{block_size} + 1$ viele Blöcke.

Hieran sieht man, dass es verschiedene formale Sichten gibt, von denen keine die andere eindeutig dominiert. Indem wir beide Sichten explizit machen, stellen wir schon auf dem Papier sicher, dass die Entwicklung in die richtige Richtung geht; und indem wir die Invarianten im Programm notieren, werden sie nicht nur automatisch geprüft, sondern bieten auch eine unermessliche Hilfestellung bei der weiteren Programmentwicklung und bei späteren Anpassungen. Wir können uns wieder die Routine zur Größenänderung anschauen, die sich direkt aus der Invariante ableitet:

```
set_size( s : INTEGER ) is
local
  new_block_size : INTEGER
do
  from new_block_size := block_size
  until s <= new_block_size*new_block_size
    -- '1 <= block_size' holds anyway.
  loop
    new_block_size := new_block_size * 2
  end
  from
  until new_block_size*new_block_size < (s * 16).max(2)
  loop
    new_block_size := new_block_size // 2
  end

  if new_block_size /= block_size
  then
    blocks := copy_blocks(size.min(s), new_block_size)
    block_size := new_block_size
  end
  size := s
end -- set_size
```

Die beiden Schleifen am Anfang verdoppeln oder halbieren `block_size` so lange, bis die Bedingung (*) wieder eingehalten wird. Das Kopieren der Array-Elemente in die neue Repräsentation erfordert gleichzeitig das Allokieren neuer Blöcke und deren Speichern in der zurückgegebenen neuen Liste, daher erledigen wir es in einer eigenen Routine. (Und da `copy_blocks` eine Funktion ist, beschreibt ihr Kopf-Kommentar das Ergebnis (den Rückgabewert) und nicht “was sie tut”.)

```
copy_blocks( s, new_block_size : INTEGER
            ) : CIRCULAR_ARRAY_LIST[BOB_NATIVE_ARRAY[T]] is
-- Liste von Blöcken der Größe ‘new_block_size’, die die Elemente ‘1..s’
-- der aktuellen Repräsentation enthält.
local
  i : INTEGER
do
  create Result
  from
    i := 0
  invariant
    0 <= i and i <= s
    Result.size - 1 = (i - 1) div new_block_size
    -- Und ‘i’ Elemente wurden schon kopiert.
  until
    s = i
  loop
    if mod(i, new_block_size) = 0
    then Result.add_last( create BOB_NATIVE_ARRAY[T]
                        .make(new_block_size) )
    end
    Result.last.put(i mod new_block_size, at(i+1))
    i := i + 1
  end
end -- copy_blocks
```

Auf die Elemente in der alten (noch aktuellen) Repräsentation können wir einfach mit der Routine `at` zugreifen. Da bereits i Elemente kopiert wurden, hat das nächste zu kopierende Element die Nummer i bei Zählung ab Null (verwendet in der Repräsentation) und die Nummer $i + 1$ bei Zählung ab Eins (verwendet an der Schnittstelle, also auch in `at`). Die Schleifen-Invariante stellt sicher, dass immer die richtige Anzahl von Blöcken allokiert ist, so dass das nächste Element stets in den letzten Block der Liste kopiert wird. So ist auch automatisch sichergestellt, dass am Ende die richtige Anzahl Blöcke allokiert wurde.

Die korrekte Implementierung der gezeigten Routinen ist nicht trivial, da man viele Gelegenheiten hat, ein “+1” oder “-1” zu vergessen oder hinzu zu zaubern. Aber mit Invarianten ist die Aufgabe beherrschbar, und wie man am vollständigen Code in [46] sehen kann, sind die restlichen Routinen der Datenstruktur dann alle einfacher als der hier vorgestellte Kern.

6.6 Zu erben ist besser als zu kaufen

Normale objektorientierte Programmiersprachen kennen nur einen Mechanismus von Vererbung (und das ist auch gut so), aber aus methodischen Gesichtspunkten ist es praktisch, zwischen zwei Aspekten zu trennen. Wir können unter *interner Vererbung* alle Aspekte zusammenfassen, die nur die erbende Klasse selbst betreffen, und unter *externer Vererbung* alle Aspekte, die auch Kunden dieser Klasse betreffen. Die interne Vererbung ist dann ein einfacher Importmechanismus, den man als Rivalen zur KUNDE-ANBIETER-BEZIEHUNG auffassen kann. Die externe Vererbung hingegen ist wesentlich schwieriger zu fassen und als Komplement zur KUNDE-ANBIETER-BEZIEHUNG zu verstehen. Die externe Vererbung umfasst insbesondere den Polymorphismus, der es einem Kunden erlaubt mit verschiedenen Anbietern dynamisch über die selbe Schnittstelle zu kommunizieren. Aber wir wollen das Einfachste zuerst behandeln...

Vom internen Standpunkt führt Vererbung im einfachsten Fall zu einem Import von Klassenelementen genau wie der Import in HASKELL oder MODULA 22 (allgemein Sprachen in denen die Module keine Klassen sind). Die erbende Klasse erhält all die Routinen der vererbenden Klasse und sie erhält auch all ihre Attribute. Die wesentlichen Unterschiede sind:

1. Die geerbten Elemente erhalten unqualifizierte Namen, sie fallen also in den Namensraum der erbenden Klasse. Ebenso fallen die Zustandsräume aller geerbten Klassen mit dem der erbenden Klasse zusammen.
2. Bei der Vererbung werden keine Elemente versteckt, es gibt also keine Datenabstraktion. Die einzige Abstraktion ist die durch Routinen.
3. Die erbende Klasse kann auch nur Spezifikationen erben und die Implementierung selbst bereitstellen, oder sie kann bereitgestellte Implementierungen überschreiben, wobei man automatisch auch alle Routinen ändert, die die überschriebene Routine aufrufen. Dies ist ein mächtiger Mechanismus, mit dem wir uns noch ausführlich beschäftigen werden.

Das automatische Erben von Spezifikationen ist die Grundlage für die externe Vererbung. Es stellt sicher, dass die erbende Klasse immer kompatibel mit der vererbenden Klasse bleibt — so kann anderer Code die Schnittstelle nutzen ohne genau zu wissen, von welcher Klasse sie eigentlich implementiert wird. Man sagt die Klassen sind wechselseitig *ersetzbar*.

6.6.1 Vererbung als Verfeinerung

Um die Ersetzbarkeit zu erreichen, müssen wir garantieren, dass jeder Kunde einer Klasse A auch mit allen Erben B von A korrekt funktioniert. Wenn wir annehmen, dass sich Kunden wirklich nur auf Angaben in den Schnittstellen ihrer Anbieter verlassen, dann reicht es schon aus, dass der Erbe B stets die selbe (oder eine stärkere) Schnittstelle als A hat. Diese Sichtweise ignoriert natürlich völlig, dass verschiedene Klassen ja erstellt werden, gerade *weil* sie etwas Verschiedenes tun! Wenn aber alle Erben von A die gleiche Schnittstelle haben wie A, dann wären alle ihre Unterschiede undokumentiert. Wie man damit fertig wird, klären wir im nächsten Unterabschnitt. Für diesen Abschnitt sei an das Prinzip von “Konsistenz statt Korrektheit” erinnert: wir nehmen in Kauf, dass die formalen Spezifikationen unvollständig sind, weil sie trotzdem schon eine große Klasse von Fehler erkennen und vermeiden helfen. Insbesondere bei der Vererbung, wo verschiedene Anbieter zusammenarbeiten müssen und man nicht mehr Spezifikation und Implementierung in einer einzigen Ansicht der Routine zusammengefasst hat, ist ein formaler Rahmen hilfreich.

Unsere bisherige formale Theorie sagt aber noch gar nichts über die “Stärke” von Schnittstellen, denn bisher haben wir ja eine Ordnung unter Programmen durch einfache Implikation hergestellt (weil Programm(fragment)e nur Bool’sche Ausdrücke sind). Schnittstellen sind aber zusammengesetzte Objekte aus benannten Elementen, so dass wir erst einmal definieren müssen, was eine “stärkere Spezifikation” überhaupt sein soll. Und um überhaupt zwei Klassenspezifikationen vergleichen zu können, müssen deren Signaturen übereinstimmen, das heißt dass eine

potenziell stärkere Spezifikation alle (exportierten) Elemente (Anfragen und Befehle) der Vergleichsspezifikation enthält. Diese Elemente müssen in beiden Klassen die selbe Signatur haben, das heißt die selben Argumenttypen und für Anfragen auch den selben Ergebnistyp.

Und wenn die Signaturen übereinstimmen, können wir zwei Klassenschnittstellen ganz einfach vergleichen: jede einzelne Routinenspezifikation des Erben B muss eine Verfeinerung der entsprechenden Routinenspezifikation aus der Klasse A sein. Außerdem kann B auch noch ganz neue Routinen enthalten, die in A nicht vorkommen. Diese Art von Verfeinerung ist also eine reine Relation zwischen Klassenschnittstellen (Spezifikationen), die Implementierungen werden nicht beachtet, denn wir wollen ja nur sicherstellen, dass zwei Klassen aus Kundensicht kompatibel sind. Für die Kunden ist auch nur der Teil der Klasseninvariante relevant, der für sie sichtbare Variablen einschränkt.

Mit dieser Definition nimmt unsere Methode einen eher operationalen Standpunkt ein: eine erbende Klasse darf alle geerbten Operationen beliebig verändern und neue hinzufügen, solange die Verträge aller exportierten Elemente (Routinen und Attribute) eingehalten werden. Die Klasseninvariante wird dabei als Spezifikation der exportierten Elemente (Anfragen) angesehen, die sie einschränkt. (Natürlich muss man sich auch an die Verträge nicht-exportierter Elemente halten, es sein denn man überschreibt alle ihre Nutzer innerhalb der Klasse.) Insbesondere kann also eine vererbende Klasse nach unserer Methode nicht festlegen, was sie *nicht* tut. Zum Beispiel kann eine Klasse nicht spezifizieren, dass ein Attribut immer konstant bleibt, es sei denn alle Befehle enthalten dies explizit als Nachbedingung. Dann kann aber eine erbende Klasse immer noch einen neuen Befehl hinzufügen, der das Attribut ändert. Andere Methoden verbieten so eine Erweiterung durch Vererbung, obwohl diese die Ersetzbarkeit von Klassen nicht im Geringsten beeinträchtigt.

Ich muss aber zugeben, dass es mindestens ein Muster gibt, bei dem eine schärfere Art der Verfeinerung, die auch Nicht-Veränderlichkeit spezifiziert, angebracht erscheint. Wenn man nämlich Objekte in eine indizierte Datenstruktur packt, so verlässt sich diese Struktur darauf, dass die Schlüssel der Objekte (bestimmte Attribute oder Hash-Werte) sich nicht ändern, solange ein Objekt in der Datenstruktur gespeichert ist. Wenn eine Klasse dies für ihre Objekte garantiert, so funktioniert die Datenstruktur reibungslos, egal wie andere Programmteile die gespeicherten Objekte verändern. Wenn man dann aber Objekte einer Subklasse in der Datenstruktur speichert, die Veränderungen des Schlüssels zulässt, ist diese Garantie nicht mehr gegeben.

In unserer Methode wird dieser Konflikt einfach durch einen Wechsel der Sichtweise gelöst: nicht die Klasse der gespeicherten Objekte ist für die Aufrechterhaltung der Schlüssel-Konstanz verantwortlich (denn bei Objekten der Klasse, die nicht in einer Struktur gespeichert sind, kann man den Schlüssel ja gefahrlos verändern), sondern die Programmteile, die die gespeicherten Objekte sonst noch verwenden, und damit in letzter Instanz der Programmteil, der die Struktur enthält und Objekte hinzufügt. Auch wenn diese geänderte Verantwortungszuweisung nur aussieht wie eine Ansichtssache, so erlaubt sie uns doch bei der eben vorgestellten, sehr einfachen Definition von Verfeinerung zu bleiben.

Leider ist die neue Sicht für Programmierer praktisch wenig hilfreich, da es für sie sehr schwer ist, alle in indizierten Strukturen gespeicherten Objekte zu verfolgen, um sicherzustellen, dass deren Schlüssel nie geändert werden. Es stellt sich also heraus, dass man in der Praxis eine Kombination beider Sichten einsetzen kann, um das Problem unter Kontrolle zu bekommen. Man verwendet dazu erstens für die gespeicherten Struktur-gespeicherten Objekte eine Klasse, die die Veränderung des Schlüssels nicht zulässt. Und zweitens lässt man den Struktur-benutzenden Code absichern, dass alle gespeicherten Objekte genau von dieser Klasse erstellt wurden und nicht von einer Unterklasse. Damit ist für alle anderen Programmteile, die diese Objekte verwenden, automatisch sichergestellt, dass der Schlüssel nicht verändert wird. Natürlich kann man trotzdem noch Unterklassen verwenden, die auch den Schlüssel nicht verändern, denn der Struktur-verwaltende Code kann ja auch Objekte dieser Unterklassen zulassen, solange keine Objekte anderer Unterklassen dazwischen schlüpfen.

Der Programmierer muss dann nicht mehr alle Programmteile überwachen, die Struktur-gespeicherte Objekte verändern, sondern nur noch diejenigen, welche solche Objekte erstellen. Weil man aber Konstruktor-Aufrufe sowieso immer minimiert und ausfaktoriert, ist dies jetzt viel einfacher.

6.6.2 Vererbung und Dynamische Bindung

“Durch die Wiederverwendung von Prozeduren kann neuer Code alten Code aufrufen. Aber erst dynamische Bindung macht es auch möglich, dass alter Code neuen Code aufrufen kann.” — Dieser Werbespruch für die Objekt-Orientierung spielt darauf an, dass man ein Programm erweitern kann, indem man eine Unterklasse zu einer vorhandenen Klasse hinzufügt. Die Kunden dieser Klasse (alter Code) rufen dann die überschriebenen Routinen der Unterklasse auf (neuer Code). Sandro Schlaumeier wird natürlich sofort einwerfen, dass man über Funktionsparameter natürlich auch ohne Objekte neuen Code von altem rufen lassen kann. Wir werden gleich sehen, dass diese Bemerkung sogar sehr tief Sinnig ist!

Zuvor muss aber die Denkweise des “alter Code ruft neuen Code” noch in ihre Schranken verwiesen werden. Denn leider wird dieses Muster des Erweiterns von Programmen durch Vererbung und Überschreiben von Routinen meist schrecklich überstrapaziert und führt dann zu ungeheuerlichen Strukturen der Software. Demgegenüber hebt die EIFFEL-Methode stets die Bedeutung der dynamischen Bindung als eine Art umstrukturierte Fallunterscheidung vor: jeder Aufruf einer polymorphen Routine verzweigt zur angebrachten Implementierung. Die Sichtweise benutzt ein Bild der Vererbung mit mindestens zwei Erben, bei der die vererbende Klasse nicht eine Vorlage ist, die man nach Belieben überschreibt, sondern eher die Ansammlung aller Gemeinsamkeiten der erbenden Klassen. In einem guten Software-Entwurf kommt Vererbung nicht zu Stande, weil neue Funktionalität durch Unterklassen hinzufügt, sondern weil man explizit Gemeinsamkeiten in Schnittstellen und Implementierungen ausfaktoriert.

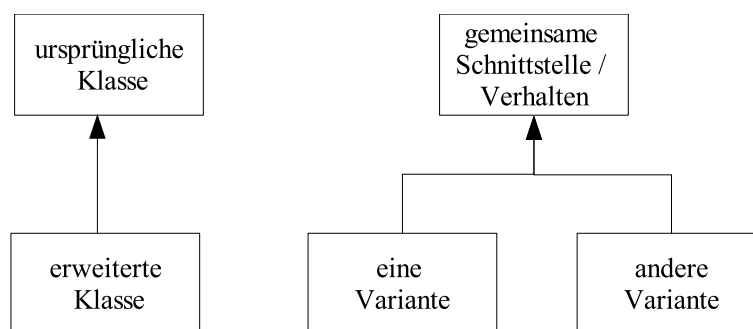


Abbildung 6.4: Vererbung zur Ergänzung von Software oder zur Schaffung von Struktur?

Auch wenn die “Interfaces” von JAVA sonst ein unnützes Konstrukt sind (wenn man keine Phobie vor Mehrfachvererbung hat, reichen auch abstrakte Klassen), so haben sie wenigstens dazu geführt, dass das Muster der gleichberechtigten Erben sich im Gegensatz zum wilden Überschreiben von Routinen durchgesetzt hat.

Dynamische Bindung als eine Art Fallunterscheidung anzusehen, ist nicht nur von methodischem Vorteil, sondern es stellt auch eine Art dar, polymorphe Aufrufe formal zu verstehen. Denn formal gesehen erhält man ein äquivalentes Programm, wenn man den polymorphen Aufruf einfach durch eine Fallunterscheidung über den Objekt-Typ ersetzt, mit den verschiedenen Routinenimplementierungen als Körper der Fälle. In der Praxis verwendet man diese Äquivalenz auch zum Refaktorisieren (hier im Abschnitt 7.3.4).

Ganz zufrieden stellend ist diese formale Charakterisierung der dynamischen Bindung aber noch nicht, denn sie setzt ja voraus, dass man alle Klassen kennt, deren Objekte möglicherweise einmal Ziel eines bestimmten polymorphen Aufrufs sind. Aber es gehört ja gerade zur besonderen Erweiterbarkeit von polymorphem Code (also solchem, der polymorphe Aufrufe enthält), dass er auch mit Objekten neuer Klassen funktioniert. Wenn man also eine Bedeutung haben will, die von all den möglichen Erweiterungen unabhängig ist, so muss man diese Klassen als Parameter für den betrachteten Code aufnehmen. Man stellt sich vor, dass die Routinen auch Attribute eines Objektes sind, und polymorpher Code ruft dann den Wert dieser Parameter auf. Vorhin

hatte ich erwähnt, dass durch Funktionen als Parameter auch “alter Code neuen Code rufen” kann; hier sehen wir, dass auch die formale Semantik der dynamischen Bindung dieses Konzept benutzt: jedes Objekt “bringt” sozusagen “seine Funktionen mit”, die Funktionsparameter werden implizit mit jedem Objekt übergeben.

Interessanter wird die dynamische Bindung auch genau auf diese Weise implementiert! Und was noch verblüffender ist: selbst die andere Sichtweise von polymorphen Aufrufen als Fallunterscheidung wird von manchen optimierenden Compilern benutzt. Beide Varianten spielen also eine Rolle sowohl in der Methode, der Theorie als auch der Implementierung!

<pre> deferred class STELLE feature brutto_gehalt : EURO_CENT gehalt : EURO_CENT is -- (Immernoch vor Steuern.) deferred end ... end class MANAGER inherit STELLE feature bonus : EURO_CENT is [...] gehalt : EURO_CENT is do Result := brutto_gehalt + bonus end ... end class INGENIEUR inherit STELLE feature gehalt : EURO_CENT is do Result := brutto_gehalt end ... end </pre>	<pre> deferred class ANGESTELLTER inherit POLITISCHE_KONSTANTEN -- 'steuersatz' feature stelle : STELLE netto_gehalt : EURO_CENT is -- in (Euro) Cents deferred end ... end class TEILZEIT_ANGESTELLTER inherit ANGESTELLTER feature netto_gehalt : EURO_CENT is do Result := stelle.gehalt * (1 - steuersatz) end ... end class VOLLZEIT_ANGESTELLTER inherit ANGESTELLTER feature fahrtkostenzuschuss : EURO_CENT is [...] netto_gehalt : EURO_CENT is do Result := stelle.gehalt * (1 - steuersatz) + fahrtkostenzuschuss end ... end </pre>
--	---

Abbildung 6.5: Zwei mal Zwei macht Vier: Gehaltsberechnung mit dynamischer Bindung.

Für das obligatorische Beispiel zu diesem Abschnitt habe ich mir etwas ausgedacht, an dem man sich auch gleich ein bisschen Mehrfachvererbung, Delegation und Mehrfachzuteilung (*multiple dispatch*) anschauen kann. Wir nehmen ein paar fiktive Formeln zur Berechnung des Netto-

Gehalts aus dem Brutto-Gehalt. Welche Formel man verwenden muss, hängt vom Angestellten (Manager oder Ingenieur) und von der Stelle ab (Teilzeit oder Vollzeit). Um es gleich noch etwas schwieriger zu machen, werden einige Zusätze vor Abzug der Steuern drangenommen, und andere danach.

	INGENIEUR	MANAGER
TEILZEIT	brutto * (1 - steuersatz)	(brutto + bonus) * (1 - steuersatz)
VOLLZEIT	brutto * (1 - steuersatz) + fahrtkostenzuschuss	(brutto + bonus) * (1 - steuersatz) + fahrtkostenzuschuss

Anfänger kommen häufig auf die Idee, diese Situation mit Mehrfachvererbung abzubilden, damit schießt man aber meist übers Ziel hinaus. Man bräuchte ja die Klassen `TEILZEIT_INGENIEUR`, `VOLLZEIT_INGENIEUR`, `TEILZEIT_MANAGER` und `VOLLZEIT_MANAGER` und dazu noch die Klassen, die man in der anderen Lösung auch noch braucht (siehe nächster Absatz).

Die wichtige Grunderkenntnis für einen besseren Entwurf zu diesem Beispiel ist, dass die Stellen- und Angestellten-Ebenen jeweils unabhängig sind. Manager unterscheiden sich von Ingenieuren durch ihren `bonus` und Vollzeit- von Teilzeit-Angestellten durch den `fahrtkostenzuschuss`. Anstelle der Mehrfachvererbung können wir also zwei getrennte Klassenhierarchien erstellen, die über eine Schnittstelle verbunden sind. Damit werden Stellen und Angestellte zur Laufzeit kombiniert (wodurch sich die Beziehungen auch zur Laufzeit ändern können). Diese Lösung ist in Abbildung 6.5 dargestellt; die Schnittstelle zwischen den beiden Hierarchien ist die Anfrage `gehalt` der Klasse `STELLE`.

Wenn man Vererbung durch die `KUNDE-ANBIETER-BEZIEHUNG` ersetzt, spricht man oft von “Delegation”, weil dann die eine Klasse des Duos einen Teil der Funktionalität an die andere weiter delegiert.

Weil bei der dynamischen Bindung nur die Klasse des Zielobjektes eines Aufrufs darüber entscheidet, welche Routine aufgerufen wird, spricht man hier von einfacher Zuteilung (*single dispatch*). Die Befürworter von Sprache mit vielen Konstrukten würden uns für obiges Beispiel vielleicht die Mehrfachzuteilung (*multiple dispatch*) empfehlen: die Routine `netto_gehalt` würde dann in vier Versionen implementiert, für jede Kombination von Klassen der beiden Hierarchien. Der Vorteil dieses Verfahrens sei eine “natürlichere Modellierung”. In der Tat spart man sich dabei die Definition einer Schnittstelle zwischen den beiden Hierarchien. Aber genau das ist auch der Nachteil, denn wenn nicht jede Routine mehr genau einer Klasse zugeordnet ist, können wir auch keine Informationen mehr verstecken, die Abstraktion geht verloren. Mehr dazu im Abschnitt 7.3.4.

6.6.3 Vererbung und Statische Bindung

Wie wir gesehen haben, gehört es zur EIFFEL-Methode, dass Spezifikationen und Code immer beisammen gehalten werden. Die Form einer in ihren Vertrag eingehüllten Routine steht exemplarisch für dieses Prinzip, und die automatische Generierung der Schnittstelle ist ein wichtiger Teil der Methode: nur weil alle Informationen zur Klasse in der selben Datei verwaltet werden, ist es leicht sie wechselseitig auf dem neuesten Stand zu halten. Wenn man aber einmal eine einzige Schnittstelle für verschiedene Klassen verwenden möchte, ist es trotzdem nützlich die Spezifikation in einer separaten Datei zu halten. Das folgende Muster zeigt, wie man Vererbung dazu benutzen kann.

Vererbung und Dynamische Bindung gehören natürlich unzertrennlich zusammen, aber oft zieht man auch ohne Dynamische Bindung Nutzen aus der Vererbung. Dieser Abschnitt zeigt ein Beispiel dafür und gibt gleichzeitig ein Muster, um die Angelegenheiten “Portabilität” und “Varianten” zu behandeln — Dinge für die C-Programmierer (und deren Erben) wohl den Präprozessor einsetzen würden. (Ein Programm, das den Programmcode vor dem Compiler bearbeitet und sich dabei um Dinge wie `#ifdefs` kümmert.) Sowohl die “Objektorientierte Methode” nach Bertrand Meyer als auch das “Professionelle Programmieren” nach Will zeichnen sich aber durch

den Verzicht auf Präprozessoren aus. Präprozessoren erschweren die Bearbeitung des Quellcodes durch andere Programme (in unserer Zeit sind hier speziell Refaktorisierungsprogramme zu nennen), und sie stellen eine weitere Komplexität des Lösungsraumes dar, die bei guter Methode und Werkzeugen einfach nicht nötig ist. Meyer [29] beschreibt, wie die Programmiersprache Gewinn bringend Aufgaben des Präprozessors übernehmen kann:

- `#includes` ersetzt man durch Module,
- Makros ersetzt man durch *inlining* von Routinen (und trennt dabei die Angelegenheit Effizienz von der Funktionalität),

insbesondere `assert` durch eine umfassende Behandlung von Verträgen und Zusicherungen,

- `#ifdefs` ersetzt man durch die normale Fallunterunterscheidung der Programmiersprache `if ... then ... else ... end`, und man verlässt sich darauf, dass der Compiler zur Übersetzungszeit bekannte Bedingungen auswertet und dann die Fallunterscheidung durch den korrekten Fall ersetzt, wobei der andere Fall als so genannter “toter Code” entfernt wird.

insbesondere `#ifdefs` zum Kennzeichnen von Test-Code, ersetzt man durch ein spezielles `debug`-Konstrukt. (Letzlich ist die parametrisierbare Variante dessen in EIFFEL schon wieder fast so ausdrucksstark wie die allgemeinen `#ifdefs`, so dass man wieder etwas Disziplin braucht, um es nicht auch noch für andere Zwecke einzusetzen.)

In diesem Abschnitt werden wir sehen, wie Vererbung eingesetzt werden kann, um einen weiteren Anwendungsfall der Präprozessoren zu behandeln (und ihre Wertlosigkeit damit zu bekräftigen).

Außerdem erfordern Präprozessoren, dass Programme als Text repräsentiert sind. Diese Betrachtungsweise stammt aus den 70ern und ist besonders nützlich im Zusammenhang mit der UNIX Werkzeugkiste für Programmierer: einer Sammlung von Programmen, die just diesen simplen Text als gemeinsames Datenformat hatten. Im Zeitalter von graphischen Entwicklungsumgebungen und XML scheint diese Idee schlicht überholt.

Portabilität erreicht man im Idealfall dadurch, dass man nur portable Funktionalitäten der Laufzeitumgebung benutzt. Wenn man portable Werkzeuge zur Verfügung hat, ist das automatisch gegeben. Meistens muss man aber auch noch eine gewisse Disziplin anwenden, um nicht “Plattform-abhängige” oder “Produkt-spezifische” Funktionalitäten zu verwenden, die fast jede Implementierung eines auch noch so portablen Standards anbietet. Oftmals reicht aber auch das nicht, und man braucht Funktionalitäten, die auf verschiedenen Umgebungen verschieden zu verwenden sind. Nach alter Methode benutzt man dazu `#ifdefs`, nach neuer Methode benutzt man einfach `if` und Konstantenauswertung des Übersetzers. Die Verfechter des Präprozessors führen dagegen folgende Nachteile an:

- `#ifdefs` an Stelle von `if` haben einen Dokumentationseffekt, weil sie zwischen Plattform-Unterscheidungen und Anwendungs-Fallunterscheidungen unterscheiden. Eine Art Trennung der Angelegenheiten.
- Man kann sich nie so richtig sicher sein, ob der Compiler wirklich allen unnützen Code entfernt und möchte das lieber genauer kontrollieren.
- Man kann sich damit nicht gegen Unverträglichkeiten verschiedener Compiler selbst absichern. (Das kommt natürlich eher selten vor, und man sollte sich dann schon genau überlegen, ob man nicht mit dem kleinsten gemeinsamen Nenner auskommt.)

Ich habe dem noch einen Nachteil beider *if*-Ansätze hinzuzufügen: die Plattform-Unterscheidungen werden über den gesamten Code verstreut; selbst wenn sie syntaktisch von Anwendungs-Fallunterscheidungen getrennt sind, ist das noch eine Vermischung von Angelegenheiten. Außerdem sind die Plattform-Unterscheidungen in gewisser Weise nur eine Flickschusterei für ein Problem, dass gar nicht existieren sollte.

Ähnliche Angelegenheiten treten auf, wenn man ein Produkt in verschiedenen Varianten liefern möchte, sei es die *light*- und *gold*-Variante oder die *Familien*- und *Geschäfts*-Variante. Das Problem ist dann, dass einige Klassen für jede Variante verschieden sind und andere sind gleich.

Falls es Klassen gibt, die Varianten-abhängige und -unabhängige Teile vermischen, so kann man die Gemeinsamkeiten leicht durch Vererbung “nach oben” faktorisieren.

Und man möchte jetzt natürlich nicht durch die Varianten-unabhängigen Klassen zusätzliche Parameter “hindurchreichen”, die die Variante bestimmen. Natürlich könnte man dynamische Bindung verwenden, so dass die Varianten-Information im Typ einiger Objekte gespeichert wird, die jeweils zur Klasse einer Variante gehören. Man hat dann zum Beispiel die Klassen `WINDOWS_FILE` und `UNIX_FILE`, die größtenteils über ihren gemeinsamen Vorfahren `FILE` benutzt werden. Das hat aber folgende Nachteile:

- Auch wenn dann die meisten Kunden dieser Klassen von den verschiedenen Varianten völlig unabhängig sind, müssen ja an einigen Stellen auch die Objekte erstellt werden, und zumindest dort braucht man wieder Informationen über die aktuelle Variante.
- Außerdem kann es ja sein, dass manche Klassen einer Variante gar nicht übersetzbar sind, wenn für eine andere Variante übersetzt wird. Diese Klassen gehören aber zum System (sie werden ja im Quellcode von zumindest den Objekt-erstellenden Teilen erwähnt) und verhindern daher dessen Übersetzung. (Dies tritt auf, wenn die Varianten für unterschiedliche Varianten einer Programmiersprache geschrieben sind. Das kann aber u.a. nötig sein, weil eine Programmiersprache nicht für alle Betriebssysteme in der gleichen Variante vorliegt.)
- Schließlich tragen bei diesem System alle Varianten noch den übersetzten Code aller anderen Varianten mit sich herum (es sei denn der Compiler optimiert wirklich sehr clever). Und das ist im Allgemeinen ziemliche Verschwendung und im Speziellen ganz sinnwidrig: wenn nämlich eine Variante die “besonders Ressourcen-schonende” ist.

Man möchte also in jeder Variante nur die Klassen haben, die zur Variante gehören; ganz so als ob die Klasse überhaupt nur in einer Variante existiert! Und das geht ganz einfach: man gibt den sich entsprechenden Klassen verschiedener Varianten einfach den gleichen Namen und hält sie dafür in verschiedenen Verzeichnissen (oder Paketen oder Zweigen eines Versionsverwalters, ...). An Stelle von `WINDOWS_FILE` und `UNIX_FILE` haben wir dann zweimal `PLATFORM_FILE`. Dann kann man zur Übersetzungszeit eine Klasse auswählen (z.B. durch Einstellung der Such-Pfade oder -Pakete oder -Zweige, ...), ohne dass die andere überhaupt vom Compiler bemerkt wird! Auch im restlichen Programmcode kann man alle Varianten der Klasse wie eine einzige benutzen; man braucht nicht einmal mehr die Fallunterscheidung bei der Objekt-Erstellung!

Durch die Vererbung wird aber trotzdem noch sichergestellt, dass all die Klassen verschiedener Varianten die selbe Schnittstelle haben. (Obwohl sie nie gleichzeitig vom Compiler gelesen werden!)

Auf diese Weise kombiniert man die Trennung der Angelegenheiten (Vorteil dynamischer Bindung) mit einer statischen Methode (weniger Laufzeit-Aufwand, kleinere Programmdateien, breitere Anwendungsmöglichkeiten).

7 Entwurf durch Alternativen

Ein wesentlicher Aspekt der Softwaretechnik ist es, Entscheidungen zu treffen, und entscheidend für den Erfolg ist die *Reihenfolge der Entscheidungen* (ein netter Artikel dazu ist David Parnas' [38]). In der Spezifikation geht es darum, zuerst über das "von außen sichtbare" zu entscheiden (früher sagte man, das "Was") und den Rest (das Innere, das "Wie") der Konstruktion zu überlassen. Der Software-Entwurf soll grundlegende Entscheidungen über das Innere, die Implementierung, fixieren. Der Programmentwurf und seine Dokumentation entscheiden wesentlich darüber, die oben genannten Kriterien für lesbare Programme zu erreichen. Deswegen spielen hier auch Kommentare und informale Erklärungen die größte Rolle. Bevor ich endlich beschreibe, wie das konkret aussieht, kommt in diesem Abschnitt letztmalig noch abstrakte Theorie.

Der Entwurf gibt die *Struktur* einer Implementierung wieder und begründet sie. Er beschreibt also Aspekte einer Beschreibung (der Implementierung) und ist daher ein besonders abstraktes Dokument. Glücklicherweise gibt es aber noch eine andere Sicht der Dinge: Der Entwurf beschreibt die Schnittstellen zwischen den Modulen und da diese auch zur Implementierung gehören, ist der Entwurf auch ein *Teil* der Implementierung. Bertrand Meyer hat diese Sicht in EIFFEL verewigt, die er als Entwurfs- *und* Programmiersprache bezeichnet.

Der Entwurf beschreibt zunächst, wie man *vorhat* die Software zu implementieren; und danach zu der *fertigen* Software stellen die Entwurfsdokumente eine Erklärung dar: wie funktioniert es und warum funktioniert es?

In der Tat kann man sich diese "Erklärung" wie eine Argumentation für die Korrektheit des Programms vorstellen — allerdings auf sehr abstraktem Niveau und nicht mit einem formalen Korrektheitsbeweis der Dijkstra'schen Art zu vergleichen. EIFFELs *Zusicherungen* unterstützen diese Dokumentation auf besondere Weise, man kann aber mit etwas Aufwand die selben Techniken auch in anderen Programmiersprachen verwenden.

7.1 Entwurf formal gesehen

Die von uns verwendete Theorie der Verfeinerung sagt ja, dass Spezifikationen und Implementierungen in der selben Sprache ausgedrückt werden (nämliche als Bool'sche Ausdrücke), nur das in der Programmierung viel weniger Notation und Begriffe zur Verfügung stehen als in der Spezifikation (und dadurch auch weniger Möglichkeiten neue Begriffe zu bilden). Man mag sich fragen, wo in diesem Modell die Software-Architektur und der Entwurf Platz haben. Hier kommt meine Erklärung exklusiv und bisher unveröffentlicht.

Zunächst einmal sollten wir uns klar machen, dass Architektur und Entwurf natürlich auch Bool'sche Ausdrücke sein sollten mit der selben Semantik wie Spezifikation und Implementierung. Natürlich kann man ganz andere Notationen verwenden, aber es muss möglich sein, alle "formal-funktionalen" Dokumente mit gewöhnlicher logischer Implikation zu vergleichen. Außerdem sollte klar sein, dass die Dokumente auf allen Stufen eine modulare *Struktur* haben sollten. Es gibt verschiedene Teile, die zu einem Ganzen verbunden werden. Und genau bei den Mechanismen der Verbindung versteckt sich der Unterschied!

Wir haben gesehen, dass das mächtigste Mittel zur Komposition von Spezifikationen die gewöhnliche logische Konjunktion ist. Wir können einfach sagen: "Die Software soll dies *und* jenes tun, *und* dann soll sie auch noch anderes tun, *und* dies möglichst schnell *und* sicher *und* so weiter *und* so fort." Wie wir im Kapitel über die formalen Theorien gesehen haben, gibt man

in der Spezifikation eines Programms praktischerweise die Funktionalität und die Laufzeit getrennt an und verlangt dann einfach, dass das Programm beide Bedingungen erfüllt (d.h. beide Aussagen impliziert). (Wenn man Michael Jacksons Problemzerlegung [22] verwendet, so erhält man verschiedene Teilprobleme, die konjunktiv verknüpft das Gesamtproblem ergeben.)

Wenn wir aber beginnen, ein Softwaresystem zu konstruieren, so wollen wir es in *Komponenten* aufteilen, die wir einzeln konstruieren können und dann einfach nur noch zusammensetzen. Die Konjunktion ist als Kompositionsoperator dann natürlich ausgeschlossen, denn wir können ja nicht ein schnelles und ein korrektes Programm einfach so zu einem schnellen und korrekten Programm zusammensetzen. Wenn man bei der Programmkomposition vorankommen will, dann solle man also zuerst die Menge der möglichen Kompositionsoperatoren einschränken. Genau das tut man ja auch beim Entwurf mit DESIGN BY CONTRACT: Die Einzelteile werden mit allen möglichen Mitteln spezifiziert, aber die Zusammensetzung findet bereits genauso statt, wie in der Programmiersprache: wir haben nur Klassen und Aufruf. Ein solcher Softwareentwurf passt also genau in das Weltbild des Bool'schen Ausdrucks mit Einschränkungen bei der Komposition!

Dokument-Typ	Notation	
	für Komponenten	für Konnektoren
Spezifikation	alles erlaubt	alles erlaubt, Konjunktion sehr beliebt
Architektur	alles erlaubt	??
Entwurf	alles erlaubt	ADTs und Routinenaufruf
Code	nur Programmiersprache	ADTs und Routinenaufruf

Nachdem sich dies so leicht geklärt hat, können wir uns die Frage stellen, ob denn auch der Begriff der "Architektur" in diesem Modell eine Entsprechung hat. Welche Art von Kompositionsoperatoren —schwächer als Konjunktion, aber stärker als Aufruf— können wir zulassen, um einen angemessenen Schritt vorwärts zu tun? Die Antwort findet sich in Texten über Software-Architektur: dort gibt es Datenrohre (*pipes*), Ereignisse, Datenbanken... all diese kann man zur Komposition von Komponenten auf Architekturebene benutzen.

Eine Datenbank wird in vielen Quellen als eigene Komponente angesehen. Vom Blickpunkt der Architektur aus gesehen, ist eine Datenbank aber durchaus ein Mittel der abstrakten Kommunikation: jede Komponente kann auf die Daten zugreifen und man kann abstrakt Datenänderungen beschreiben, ohne sich darum zu kümmern, welche Aufrufe später (im Entwurf) dann benutzt werden, um diese Änderungen auf der Datenbank auszuführen. Im Wesentlichen sind ja auch E/R-Diagramme (und Daten-Klassendiagramme) eine abstrakte Repräsentation von Datenbanken auf Architekturebene.

Diese Kompositionsoperatoren der Architektur nennt man *Konnektoren* und man kann sie benutzen, um die Interaktion der Komponenten auf Architekturebene zu beschreiben, ja sogar formal spezifizieren, wenn man möchte. Die Art der Konnektoren bestimmt den *Stil* einer Architektur, und je nach diesem Stil kann man verschiedene Notationen verwenden, um Komponenten zu spezifizieren. Bei Ereignis-orientierten Architekturen bietet sich ein Prozesskalkül an (*CSP* oder die entsprechenden Notationen von APTOP), bei Datenrohren kann man einfach Sequenzen als Kommunikationsmedium benutzen (oder auch einen Prozesskalkül, falls die Kommunikation synchron ist). (Diese Beobachtung stammt auch von mir und ist in der einschlägigen Literatur [39] nicht zu finden.)

Im nächsten Schritt des Software-Entwicklungsprozesses kann man dann entscheiden, wie man die Konnektoren implementiert. Ereignisse kann man zum Beispiel mittels der Unterbrechungen (*interrupts*) einer Echtzeit-Hardware implementieren oder über Warteschleifen oder über Rückrufe oder mit einer Programmiersprache, die direkt Ereignisse unterstützt. Für Datenrohre kann man den bei UNIX eingebauten Mechanismus verwenden (dann müssen die Komponenten in verschiedenen Prozessen laufen), man kann den Umweg über Dateien gehen (die man beliebig zwischenspeichern und transportieren kann) oder Daten über einen Netzwerkstrom schicken.

Peter Pragmatiker kommt hier nun schon wieder auf die Idee, für die Datenrohr-Architektur einfach eine Klassenbibliothek zu erstellen, so dass man die Komponenten einfach gegen

abstrakte Datenquellen und -senken spezifizieren und implementieren kann. Die Bibliothek stellt dann alle möglichen konkreten Implementierungen für diese Datenquellen und -senken bereit: von Prozess-intern über Dateien bis zum Internet-Verkehr, alles mit oder ohne Zwischenpuffer. Peter hat dann den Konnektor "Datenrohr" von einem Kompositionsmechanismus der Architektur zu einem Mechanismus der Implementierung gemacht und wieder einmal ein Werkzeug entworfen, das Teile der Methode überflüssig macht. Trotzdem kann Peters pragmatisches Werkzeug eine rein abstrakte Datenrohr-Architektur nicht ersetzen. Denn eine solche Architektur kann man ja auch ganz einfach mit `print` als Senke, `read` als Quelle und UNIX-Datenrohren als Konnektoren implementieren — dies ist natürlich wesentlich beschränkter, als die Verwendung einer abstrakten Klassenbibliothek (insbesondere für die Wiederverwendung), aber es ist ja auch so viel einfacher! Und darauf kann es im speziellen Fall ankommen.

Genauso kann uns Wolfgang Werkzeugverkäufer eine Klassenbibliothek zum abstrakten Zugriff auf Datenbanken anbieten, um auch diesen Konnektor zu operationalisieren. Und auch da würde es für manche Anwendungen ausreichen, einfach nur eine kleine Datenstruktur (einige Mengen und/oder Abbildungen) im Speicher zu halten, also gar keine Datenbank zu verwenden. Man muss dabei auch bedenken, dass eine abstrakte Datenrohr- oder Datenbank-Architektur eigentlich auch nur eine weitere Art ist, diese Konnektoren zu implementieren. Zunächst erscheinen sie als eine Art Universallösung, aber die Praxis zeigt, dass auch sie nicht universell genug sind. Dies liegt auch daran, dass bereits bei der vollständigen Formalisierung eines Konzepts bestimmte Entwurfsentscheidungen getroffen werden, die dann natürliche für alle kompatiblen Implementierungen festliegen. Für graphische Benutzeroberflächen wurden zum Beispiel schon vielerlei Klassenbibliotheken entwickelt, aber keine ist wirklich universeller als alle anderen. Die SWING Bibliothek sollte zum Beispiel als Teil von Java besonders Plattform-unabhängig sein. Trotzdem ist sie in der VGUI-Bibliothek von `sd&m` nur eine von mehreren Implementierungen. Dieses Beispiel zeigt, dass solche Klassenbibliotheken (die Konnektoren eines Architekturstils implementieren) für bestimmte Teilgebiete der Softwaretechnik nützlich sind: betriebliche Informationssysteme im Falle von `sd&m` und seiner Quasar-Architektur [26, 25]; Internet-Applets im Falle von JAVA. Die unglaubliche Universalität von Software macht eine solche Charakterisierung schwierig, weil man natürlich auch betriebliche Informationssysteme in JAVA schreiben kann, so wie viele andere Sachen, für die es zunächst gar nicht gedacht war.

Im Wesentlichen haben wir hier wieder den gleichen Konflikt wie schon zwischen Spezifikation und Implementierung: Höhere Programmiersprachen und bessere Werkzeuge bringen die Programmiersprachen immer näher an die formalen Spezifikationssprachen; dies ist einer der Gründe, warum informale Spezifikationen methodisch so wichtig sind. Potenzielle formale Architektur-Sprachen sind noch näher an der Implementierungsebene und daher noch mehr bedroht von operationalen Architektur-Methoden: immer höhere Konnektoren in Programmiersprachen und -bibliotheken. Für wirklich universelle Architektur-Dokumente bleibt also nur die informale Sprache übrig.

Aus der obigen Tabelle sehen wir, dass eine Software-Entwurfssprache gleich ist einer Sprache zur Spezifikationen von Softwareelementen (Routinen, ADTs) plus einem Modulkonzept. Letzteres muss das gleiche sein wie in der Programmiersprache, deswegen sind Entwurfs- und Programmiersprachen ja so eng verbunden. EIFFEL ist gar nur eine Sprache für beides, und falls nicht bald noch etwas besseres erfunden wird, kommt man wohl bald um die JAVA MODELING LANGUAGE (JML) nicht mehr herum. Graphische Entwurfsnotationen werden bald vielleicht endlich den Platz finden, der ihnen zusteht: als Illustrationen von richtigen Entwurfsdokumenten, automatisch von Werkzeugen generiert. (So wie das in EIFFEL schon seit den 90ern der Fall ist.)

7.2 Entwurf als Prozess

Wir wissen jetzt so ungefähr, was ein Software-Entwurf ist. Aber wie kommt man dahin? Die Literatur berichtet von verschiedensten Ansätzen: Top-Down, Bottom-Up, Funktions-orientiert, Daten-orientiert, Aspekt-orientiert. Alle geben sie aber nur grobe Hinweise und führen nicht zu

einem fertigen Entwurf. Die Methode, die ich hier vertrete, besteht darin einen Entwurf ständig anhand der *Kriterien* (aus Abschnitt 2.3) zu *bewerten* und gegebenenfalls zu verbessern. Aber warum ist das überhaupt so schwierig?

Entwurf im Allgemeinen bedeutet: mehrere Entscheidungen treffen, die miteinander verflochten sind. Diese Entscheidungen müssen so getroffen werden, dass sie einige harte Kriterien erfüllen (bei Software: Funktionalität, Erfüllung der Spezifikation) und einige weiche Kriterien möglichst optimieren (die Kriterien für innere Qualität).

Verflochtene Entscheidungen treffen, heißt: jede Entscheidung beeinflusst auch eine andere. Man kann sich also nicht die Entscheidungen alle der Reihe nach vornehmen und optimal treffen, weil man dann manche Entscheidungen schon implizit getroffen hat, bevor man explizit darüber nachgedacht hat. Diese Entscheidungen sind dann meist suboptimal getroffen worden und man möchte sie gern ändern. Dabei wird man aber auch wieder andere Entscheidungen mit ändern, usw, usf. Man kann also beim Entwerfen nicht einzelne Entscheidungen bewerten, sondern nur ganze Bündel von getroffenen Entscheidungen, also eine komplette Lösung oder einen Teil davon. Natürlich sind nicht alle Entscheidungen eines Entwurfsprozesses voneinander abhängig, die große Hoffnung ist ja, dass man Entscheidungen *hierarchisch* treffen kann — aber dies ist nur begrenzt möglich.

Schauen wir uns dazu ein Beispiel an, das sehr oft auftritt: eine Klasse (im Sinne von Modul) soll einer anderen eine komplexe Information mitteilen, deren Codierung wir nicht an der Schnittstelle zwischen den beiden Klassen sehen wollen. Früher trat das Problem schon auf, wenn man nur eine Menge oder Liste von einfachen Elementen übertragen wollte: plötzlich hatte man die fixe Größe eines Arrays oder Zeigerstrukturen mit im Haus und musste sie aus der Schnittstelle fernhalten. Heutzutage kann man in diesem Fall einen abstrakten Container in die Schnittstelle schreiben und es passt. Es gibt aber immer wieder kompliziertere Fälle, in denen man eine eigene Klasse benutzt, deren Objekte dann zwischen den beiden anderen Klassen ausgetauscht werden. Anstatt also die Schnittstelle zwischen zwei Klassen zu entwerfen, haben wir plötzlich schon drei. Und dazu kommen natürlich noch alle anderen, die die “Kommunikationsklasse” möglicherweise noch benutzen.

Man muss also wirklich mehrere Entscheidungen gemeinsam betrachten und man kann auch nicht manche der Entscheidungen schon im Vorhinein treffen:

Ein Entwurf dient immer der konkret gegebenen Aufgabenstellung. Für diese kann ein Entwurf *besser* sein als ein anderer. Es gibt aber keinen allgemein *besten* Entwurf.

Dies erklärt, warum allgemeine Diskussionen um Software-Entwurf (wie Entwurfs-Muster [9] oder Architektur-Stile [39]) nicht über Platituden hinauskommen: man kann Alternativen nur gegenüber einem konkreten Problem bewerten. Und so ein Problem hat oft überlagerte Aspekte, die es mit einer einzigen Lösung abzudecken gilt. Natürlich möchte man am liebsten jeden Aspekt in einem Modul kapseln, aber das Beispiel des Online-Buchladens sollte gezeigt haben, dass manche Aspekte übergreifend sind — und genau diese muss man im Entwurf behandeln.

7.2.1 Pragmatik von Schnittstellen

Wir haben uns im letzten Kapitel auf die Syntax und Semantik der Programmiersprache konzentriert, in diesem Kapitel kommt die Pragmatik hinzu. Wir können sagen, dass die Syntax von Schnittstellen hauptsächlich aus den *Signaturen* von Funktionen und Prozeduren besteht, das heißt deren Namen und Listen von Parametern mit deren Typ. Die Syntax legt also das Format für die Kommunikation fest: In welcher Verpackung wird die Funktionalität angeboten? Die Semantik hingegen ist eine komplette Beschreibung dieser Funktionalität und das vorige Kapitel hat sich ausführlich damit beschäftigt. Traditionelle Kommentare in und Dokumentationen von

Programmen befassen sich hauptsächlich mit der Pragmatik, also den Fragen “Wozu ist es gut?” und “Warum wurde dieser Programmteil überhaupt programmiert?” Und “Warum wurde er *so* programmiert?”

Gerade bei Programmteilen, die keine einfache Spezifikation haben und die sehr spezifisch für die Anwendung sind, zu der sie gehören, ist eine pragmatische Beschreibung angebracht. Bei so einer Angabe des Zweckes handelt es sich entweder um einen Verweis auf einen Teil der Programmspezifikation oder auf einen anderen Programmteil, der den spezifizierten Teil verwendet. Ein formaler Spezialfall solcher Dokumentation wird oft verwendet und kann sogar automatisch generiert werden: man gebe zu jeder Routine alle Aufrufer an (bzw. zu jeder Klasse alle Kunden). Die Angabe des Zwecks eines Programmteils (bzw. einer Schnittstelle) ist die *Rechtfertigung*, dass man sich überhaupt damit beschäftigt. Der Zweck muss nicht immer explizit genannt sein, weil er oft schon aus der Semantik hervorgeht (wenn man die Programmspezifikation und den umgebenden Code kennt). Aber ein Programmierer sollte sich über den Zweck immer bewusst sein.

Die Pragmatik, also die Dokumentation des Zwecks innerhalb der Anwendung, darf aber nicht mit der Semantik (Dokumentation der Funktionalität, d.h. Spezifikation) vermischt werden. Denn eine unabhängig dokumentierte Funktionalität ist Voraussetzung für eine erfolgreiche Wiederverwendung auch in anderen Umgebungen (oder anderen Teilen des selben Projektes). Wir haben im Kapitel über die Qualität schon gesehen, dass die Wiederverwendung gerade eine Befreiung vom konkreten Kontext einer Anwendung verlangt, aber gerade dieser Bezug drückt sich ja in der Zweck-Dokumentation aus. Folglich braucht man in der Regel mehr Zweck-Dokumentation für weniger wiederverwendbare Programmteile.

7.2.2 Alternativen

Ich habe hier das Stichwort *Alternativen* aufgeworfen: in anderen Disziplinen als der Softwaretechnik werden diese wirklich explizit gemacht (zum Beispiel in der Architektur von Gebäuden). Entwurf besteht darin, zunächst Alternativen zu *generieren* und alsdann Alternativen zu *eliminieren*, bis nur noch eine übrig bleibt. Meiner Erfahrung nach fehlt dabei aber ein im Softwareentwurf sehr nützlicher Punkt: wenn man Alternativen bewertet, stellt man deren Schwachstellen fest, was oft zur Idee für eine neue Alternative führt, man *iteriert* also die Alternativen.

Erste Alternativen generieren kann man:

1. Naiv, indem man jeder Angelegenheit ein Modul zuordnet und sich Schnittstellen überlegt.
2. Module aus der Spezifikation extrahieren (die ist meistens modular).
3. Einen Entwurf aus einem vorhergehenden, ähnlichen Projekt wieder verwenden.

Alternativen iterieren:

1. Schwachstelle einer Alternative bezüglich eines Kriteriums verbessern
2. zwei Alternativen mischen
3. Alternative verfeinern oder vervollständigen

Alternativen eliminieren:

1. Wenn eine andere Alternative direkt überlegen ist.
2. Wenn man zu viele hat, lässt man schlechter erscheinende weg.

Man sieht an der Formulierung schon, dass Alternativen keiner Totalordnung unterliegen. Das macht aber nichts, denn wenn keine Alternative besser ist als die andere, kann man einfach irgendeine nehmen. Für den professionellen Programmierer ist zwar nur der beste Entwurf (also der einfachste) gut genug, aber weil dieser ja nicht eindeutig bestimmt ist, braucht man nicht alle Alternativen bis ins Detail auszuspezifizieren, um sich für eine zu entscheiden.

Im Software-Entwurf besteht die Beschreibung einer Alternativen aus den Pragmatik-orientierten Kurzbeschreibungen der benötigten Klassen; bei höherem Detailgrad auch der Schnittstelle. Wenn man sich für eine Alternative entschieden hat, muss man nur noch die Schnittstellen zu Ende spezifizieren. An kritischen Stellen ist es dabei hilfreich in einem Kommentar die nächstbeste Alternative zu erwähnen und warum man sie nicht gewählt hat. Genau das ist die Begründung für einen Entwurf: nicht, dass es “der absolut beste” ist, sondern einfach, dass die Alternativen nicht so gut sind. Wie ich schon im Kapitel 2 schrieb: Lösungs-Entwürfe lassen sich eben immer nur ganz konkret pro Problemstellung vergleichen.

7.3 Alternativen in OO

In diesem Abschnitt wollen wir kurz einige der Sprachelemente aus dem letzten Kapitel gegenüber stellen, zwischen denen sich ein Programmierer oft entscheiden muss. Obwohl die Konstrukte formal gesehen äquivalent sind (also quasi-automatisch ineinander umwandelbar), führen sie jeweils zu anderen Entwürfen. Oftmals ist eine Variante strukturell einfacher, aber andererseits verfügt die etwas komplizierte Version über eine zusätzliche Abstraktionsebene (oder Indirektion), so dass sie die Erweiterung in eine bestimmte Richtung einfacher macht. Die Extremen Programmierer gehen davon aus, dass man während des Programmierens nicht mit einem festen Entwurf beginnt, sondern dass man einfach nur ständig neue Funktionalitäten zu einem Prototypen hinzufügt und zwischendurch laufend refaktoriert (restrukturiert), um den Entwurf an die Funktionalität anzupassen. Die Grundregel dabei lautet, dass man in jedem Moment immer nur eines von beiden tut: entweder die Funktionalität anpassen oder die innere Struktur ändern, wobei das Gesamt-Verhalten aber absolut gleich bleiben soll. (Dies wird u.a. durch semi-automatische Refaktorisierungsregeln und durch ständiges Testen sichergestellt. Meine Quelle dazu ist das Buch von Fowler [7].)

Zur Terminologie: Der Übersetzer von Fowlers Buch verwendet den Begriff “Refaktorisierung” für das englische *refactoring*, und dies zu recht, denn das Wort “faktorieren” insbesondere “ausfaktorieren” ist bereits für mathematische Umformungen in Gebrauch, deren Natur dem objektorientierten Faktorieren durchaus nahe steht. Man vergleiche insbesondere die mathematische Umformung $15ab + 9bc = 3b(5a + 3c)$ mit dem Ausfaktorieren von Gemeinsamkeiten zweier Klassen in eine gemeinsame Oberklasse.

Ich finde aber “refaktorisieren” etwas lang und umständlich für ein beim Programmieren so wichtiges (und zukünftig alltägliches) Konzept. Ich werde daher in dieser Arbeit “refaktoriieren”, “faktorieren”, “das Refaktoriieren” als Prozess und “die Refaktoriierung” für eine einzelne Refaktorisierungsregel verwenden.

Ich stimme überein, dass das Refaktoriieren eine der wichtigsten modernen Programmieretechniken ist, allerdings wird sie von den XP’lern (insbesondere [7]) in einem viel zu engen Licht gesehen. Anstatt den Code immer nur Zeile für Zeile zu ändern, kann man die Äquivalenzen und Alternativen nämlich schon von Anfang an benutzen, um Entwurfsentscheidungen besser zu treffen und zu dokumentieren. Man kann in den Quellcode-Kommentaren direkt auf die hier genannten Alternativen wie auf Entwurfsmuster verweisen und damit knapp und präzise eine leicht zu pflegende Dokumentation erhalten, deren Abstraktionsniveau weit über der Quellcode-Ebene liegt. In diesem Abschnitt zeige ich einige solcher Alternativen an einem Fallbeispiel auf. Die hier gegebene Darstellung ist aber nur der Anfang einer Methodik, die in der Literatur überhaupt noch nicht bekannt ist und in der nächsten Zeit hoffentlich umfassend erforscht und einem breiten Publikum zugänglich gemacht wird.

Im Programmwurf gibt es immer einige Konstrukte oder Muster, die man überhaupt nicht verwenden sollte, und einige Konstrukte, zwischen denen man je nach Situation auswählt. Für erstere bietet die Programmiermethode stets überlegene Alternativen, für letztere bietet die Methode nur Heuristiken, die die Auswahl erleichtern sollen. Fowler [7] unterscheidet nicht explizit zwischen beiden (was den Nutzen seines Werkes schmälert), aber man kann es indirekt erkennen, weil er für manche Muster nur Refaktorisierungen bietet, um sie zu entfernen, und zwischen anderen bietet er Refaktorisierungen in beide Richtungen.

Bei den hier vorgeführten Entwurfalternativen überwiegt immer die Sicht des Programmierens als Spezifizieren, die Lesbarkeit des Programms ist also entscheidend, nicht die Effizienz. Auch wenn alternative Versionen eines Programms manchmal unglaublich unterschiedlich sind, so beschreiben sie trotzdem doch genau den gleichen Algorithmus! Darin sieht man, dass die objektorientierte Struktur eigentlich nur eine redundante Schicht in der Programmierung darstellt, aber ohne sie kann man Programme nicht vernünftig gliedern.

7.3.1 Variablen, Funktionen, Ausdrücke

Die einfachsten Umformungen sind die mit Bool'schen Ausdrücken und bedingten Anweisungen. Wenn man Bool'sche Ausdrücke vereinfacht so geht man einen Kompromiss ein zwischen der Kürze des vereinfachten Ausdrucks und der Struktur des längeren Ausdrucks, die möglicherweise seine Herkunft besser widerspiegelt.

```
alles_bereit : BOOLEAN is
  do Result := a and b end
```

```
alles_klar : BOOLEAN is
  do Result := b and c end
```

Wenn man zum Beispiel diese beiden Routinen gegeben hat, dann sind die Ausdrücke `alles_bereit implies alles_klar` und `alles_bereit implies c` äquivalent.

In Fallunterscheidungen sollte es selbstverständlich sein, dass die verschiedenen Fälle keinen gemeinsamen Code enthalten. Wenn man diesen nicht vor oder nach die Fallunterscheidung verschieben kann, so sollte man ihn in eine eigene (wenn auch noch so kleine!) Routine auslagern, die dann von beiden Zweigen mit unterschiedlichen Parametern aufgerufen wird.

Spielraum bei der Gestaltung von Fallunterscheidungen hat man durch folgende Äquivalenzregeln:

```
if a then
  if b then A else B end
else ... end
≡
if a and b then A
else if a and not b then B
else ... end
```

Mit dieser Äquivalenz kann man einerseits eine Mehrfachfallunterscheidung "abflachen", so dass alle ihre Fälle gleichwertig sind, oder andererseits die Fälle in zwei Gruppen aufteilen — je nachdem was den Anforderungen näher liegt.

Bei der folgenden Äquivalenz mag die linke Seite schlecht erscheinen, weil der Teil "A" wiederholt wird. Wenn man aber eine längere Liste von Fällen hat, möchte man möglicherweise explizit betonen, dass zwei Fälle gleich behandelt werden. (Natürlich ist dabei der wiederholte Teil nur sehr kurz, wie eine einzelne Zuweisung oder ein Routinenaufruf.)

```
if a then A
else if b then A
else ... end
≡
if a or b then A
else ... end
```

In der rein funktionalen Programmierung haben wir ja überall nur Ausdrücke, und für diese gilt die Grundregel: gleiche oder ähnliche Teilausdrücke werden in Funktionen (oder lokalen Variablen) ausgelagert. In der imperativen Programmierung benutzt man oft Variablen, an die man nur einmal zuweist, um den Wert eines einzelnen Ausdrucks zu speichern (technisch) und zu benennen (methodisch). (Das heißt, die lokale Variable dient gleichzeitig der Effizienz und der Lesbarkeit.) Dies entspricht den lokalen Variablen der funktionalen Programmierung: "sei x gleich ...". Funktionen und lokale Variablen kann man oft durch einander ersetzen. Lokale Variablen bevorzugt man dabei, falls der benannte Wert von sehr vielen lokalen Parametern abhängig ist (die müsste man an eine Funktion ja alle einzeln übergeben) oder falls der benannte Wert sehr spezifisch für die aktuelle Routine ist. Funktionen bevorzugt man im entgegengesetzten Fall: wenn wenige Argumente verwendet werden und wenn der Wert der Anfrage auch in anderen Routinen interessant sein könnte.

In der imperativen Programmierung kann man ja an Variablen mehrfach Werte zuweisen, dabei gilt die Regel: jede Variable wird nur zu einem Zweck benutzt, also nicht am Anfang und am Ende einer Routine mit unterschiedlichen Werten belegt, die unabhängig voneinander benutzt werden. (Diesen Fehler kann man leicht durch Aufteilen der Variable auf zweie (mit besserem Namen) beheben.) Hier sehen wir eine ganz schlechte Routine, die eine Variable mehrfach unabhängig zuweist und in eine andere akkumuliert, und dies auch noch mit einer anderen Re vermischt:

```

movie_points : INTEGER is
local
  average : INTEGER
do
  average := professional_review_points \ num_professional_reviews
  Result := average + integer_square_root(num_professional_reviews)

  average := customer_review_points \ num_customer_reviews
  Result := Result + average + integer_square_root(num_customer_reviews)

  average := audience_voting_points \ num_audience_voting
  Result := Result + average + integer_square_root(num_audience_voting)
end

```

An diesem Beispiel kann man *sehr* viel verbessern. Wir fangen aber klein an und entfernen zuerst die Hilfsvariable. Die Ausdrücke werden zwar länger, aber das Berechnungsmuster wird offensichtlicher:

```

movie_points : INTEGER is
do
  Result := professional_review_points / num_professional_reviews
           + integer_square_root(num_professional_reviews)
  Result := Result + customer_review_points / num_customer_reviews
           + integer_square_root(num_customer_reviews)
  Result := Result + audience_voting_points / num_audience_voting
           + integer_square_root(num_audience_voting)
end

```

Das Berechnungsmuster auszufaktorisieren macht zwar den Programmcode nicht viel kürzer, aber doch etwas klarer. Und was noch viel nützlicher ist: wir können die Funktion jetzt ganz funktional schreiben, mit nur einer Zuweisung an `Result`:

```

points( a, b : INTEGER ) : INTEGER is
do
  Result := a / b + integer_square_root(b)
end

movie_points : INTEGER is
do
  Result := points(professional_review_points, num_professional_reviews)
           + points(customer_review_points      , num_customer_reviews      )
           + points(audience_voting_points    , num_audience_voting    )
end

```

(Die Namen `a`, `b` sind nicht dauerhaft gewählt und verschwinden gleich wieder.) Letztlich können wir die sechs verwendeten Variablen besser strukturieren, so dass der paarweise Zusammenhang besser ausgenutzt wird:

```

class POINTS feature
  sum, num : INTEGER

  points : INTEGER is
  do
    Result := sum / num + integer_square_root(num)
  end
end -- class POINTS

class MOVIE feature
  professional_reviews, customer_reviews, audience_voting : POINTS

  points : INTEGER is
  do
    Result := professional_reviews.points
      + customer_reviews.points
      + audience_voting.points
  end
end -- class MOVIE

```

Wie leicht dieser Code jetzt aussieht! Kompiliert man ihn, erhält man (fast) das selbe Ergebnis wie bei der ersten Version. Aber während jene eine stupide Rechenanweisung darstellt, sieht man hier, wie sich die (ausgedachte) Bewertung eines Films wirklich zusammensetzt.

Aber wir sind mit dem Siegeszug des Refaktorisierens noch lange nicht am Ende. Die Schritte, die man zur Umformung von Ausdrücken und Funktionen anwendet, sind nämlich sehr oft die gleichen, und man kann daraus sogar ein halb-automatisches Verfahren machen. Stellen Sie sich vor, ihre Entwicklungsumgebung findet zu komplizierten Code und bietet Ihnen Verbesserungsvorschläge an, die sie dann auch noch automatisch umsetzt! Hier das Verfahren:

1. Zuerst werden alle lokalen Variablen entfernt (außer denen, die in Schleifen zugewiesen werden).

Dieser Schritt macht die Routine natürlich erst einmal komplizierter; er ist auch nur notwendig um die Routine zu normalisieren, so dass man Möglichkeiten zum Faktorisieren besser erkennt. Man kann sagen, dieser Schritt entfernt zunächst die alte Struktur, bevor eine neue angewendet werden kann. In [21] wird übrigens der selbe Schritt zu theoretischen Zwecken angewandt. Möglicherweise ist es auch nützlich, einige Funktionen in die Routine auspacken (d.h. Funktionsaufrufe durch den Funktionskörper zu ersetzen). Man könnte das zum Beispiel mit allen Funktionen machen, die nur von dieser Routine benutzt werden.

Der Programmierer muss übrigens diese längere Normalform überhaupt nicht zu Gesicht bekommen, ein Refaktorisierungsprogramm kann sie intern verwenden.

2. Dann werden die Ausdrücke in der Routine auf Gemeinsamkeiten untersucht, die man in Funktionen auslagern kann.

Dies geht mit einem relativ einfach Unifikationsalgorithmus. Dem Programmierer werden dann mögliche auszulagernde Funktionen angeboten, zusammen mit einer Liste der Stellen an denen sie dann verwendet würden. Für jeden Teilausdruck, der bei verschiedenen Aufrufen unterschiedlich ist, und für jeden Teilausdruck, der von den Routinen-Parametern abhängt, wird dabei ein Funktionsargument eingeführt.

Der Programmierer kann den Funktionen und ihren Argumenten dann Namen geben und sich entscheiden, die Funktion auszulagern.

3. Danach kann man in beliebiger Reihenfolge die entstandenen Funktionen verschieben (siehe übernächster Abschnitt) und die verbleibenden gemeinsamen Teilausdrücke wieder in lokale Variablen zusammenfassen.

Das Programm merkt sich dabei die Namen ausgepackter Funktionen und Variablen, und falls genau wieder diese Funktion oder lokale Variable ausfaktoriert wird, bekommt sie wieder den alten Namen.

7.3.2 Prozeduren

Funktionen und Ausdrücke kann man in objektorientierten Programmiersprachen fast ebenso leicht umformen wie in funktionalen Sprachen. Etwas schwieriger gestaltet sich das Ausfaktorisieren von Prozeduren. Im Idealfall sollte man ja nur ein paar Zeilen Code (also Elemente einer Programm-Sequenz, es zählen ja die Semikoli nicht die Zeilenumbrüche) in eine neue Prozedur kopieren und diese innerhalb der Sequenz aufrufen. Falls die zu faktorierenden Zeilen keine lokalen Variablen oder Argumente verwenden, funktioniert das auch genau so. Falls sie auf solche Variablen nur lesend zugreifen, kann man diese als Argumente übergeben, und es funktioniert auch. Aber falls solche Variablen auch geändert werden, funktioniert das nicht mehr.

Es ist dabei interessant zu bemerken, dass die “strukturierten Programmiersprachen” wie *Pascal* auch die letzte Variante leicht faktorisieren können, indem man eine Prozedur lokal zu einer anderen macht. Dieses Konstrukt ist aber eines der wenigen (wie übrigens auch Referenz-Parameter), die man nicht in die objektorientierte Methode übernommen hat. Es gibt aber ein einfaches Muster, genannt “Routinenklasse” (“Methodenobjekt” unter XP’lern), mit dem man diesen Effekt erreichen kann: man lagert eine Routine einfach in eine neue Klasse aus, macht ihre lokalen Variablen zu Attributen dieser Klasse (und die Argumente zu unveränderlichen Attributen), und dann kann man Prozeduren aus der Routine auslagern, die auf die lokalen Variablen und Argumente zugreifen können. Dort wo die Routine ursprünglich mal war, schreibt man nur eine Zeile, um ein Objekt der neuen Routinenklasse zu erstellen und auf diesem die neue Routine aufzurufen. Dadurch bleiben Aufrufer der Routine unabhängig von dieser Art der Implementierung.

Eine kleinere Variante dessen ist, einfach einige lokale Variablen einer Routine zu Klassenattributen zu machen, die nur zur Kommunikation der Routine mit ihren Unter-Routinen benutzt werden. Noch besser ist es natürlich, wenn man es schafft, eine Routine so zu vereinfachen, dass man nur Funktionen und Befehle auf der Klasse auslagern kann.

7.3.3 Wohin mit der Routine?

Die meisten Routinen verwenden gleichzeitig Elemente mehrerer Klassen, so dass nicht unbedingt klar sein muss, in welche Klasse eine Routine genau gehört. Auch wenn man Routinen aus anderen herausfaktoriert, stellt man sie zunächst in die gleiche Klasse, obwohl vielleicht eine andere Klasse besser wäre. Zum Verschieben von Routinen zwischen Klassen erwähnt Fowler als einen Punkt, dass man der Routine in der neuen Klasse irgendwie Zugriff auf die alte Klasse verschaffen muss, weil sie ja noch einige von deren Elementen verwenden könnte. Dabei lässt er aber unter den Tisch fallen, dass die Routine natürlich auch Zugriff auf die richtige *Instanz* dieser Klasse braucht. Überhaupt muss man beim Platzieren von Routinen nicht nur über die Klasse entscheiden, sondern auch über das Objekt.

Nehmen wir zum Beispiel an, wir haben in einer Klasse *X* folgende Hilfsroutine ausfaktoriert und wir stellen fest, dass sie eigentlich in die Klasse *HTML* gehört:

```
append( a, b : HTML ) : HTML
```

Wir können die Routine jetzt so verschieben, dass *a* oder *b* das Zielobjekt der neuen Routine ist. (Man könnte die Routine auch als Konstruktor implementieren oder an einem *facility*-Objekt, dass selbst keine Daten hat, aber solche Umwege müssen wir nicht gehen.) Nehmen wir den ersten Parameter als Ziel, ändert sich ein Aufruf `append(a, b)` in `a.append(b)`.

Wenn man Routinen verschiebt, stellt man sich am besten vor, dass die Trägerklasse der Routine auch ein Parameter ist. Man kann dann eine Routine der Form `{TARGET}.routine(P1, P2)` in jede der Klassen *TARGET*, *P1*, *P2* verschieben. (In der gezeigten Form ist sie natürlich schon in der Klasse *TARGET*.)

Eine weitere Refaktorisierung ergibt sich für alle Parameter einer Routine, auf die sie nur lesend zugreift (d.h. sie ruft nur Anfragen auf, keine Befehle). Dann kann man nämlich auch die Ergebnisse dieser Anfragen direkt übergeben an Stelle des ganzen Objekts. Vorteil: man sieht direkter, welche Werte verwendet werden und die Routine wird unabhängig von der Klasse des Parameters und kann auch in anderen Kontexten (ohne diese Klasse) verwendet werden. Falls umgekehrt eine Routine zu viele Parameter hat, dann kann man die Umwandlung umgekehrt durchführen und ein Objekt übergeben, von dem sich die Routine dann die Informationen “selbst besorgt” (O-Ton Fowler). Falls kein passendes Parameterobjekt existiert, kann man auch eine neue Klasse schreiben. Der Aufrufer der Routine erstellt dann ein Objekt dieser Klasse und übergibt es. Oder er übergibt sich (sic). Ein Parameterobjekt zu übergeben macht die Signatur kürzer und die Implementierung der Routine flexibler: sie kann nun ganz ohne ihre Schnittstelle zu verändern mehr oder weniger oder andere Informationen von dem Parameterobjekt bekommen.

Man kann dies auch mit der vorigen Refaktorisierung kombinieren, zum Beispiel:

```
class KUNDE feature
  adresse : ADRESSE
  adresse_gueltig : BOOLEAN is
    do .... end
end -- class Kunde
```

Die Anfrage `adresse_gueltig` hat als Parameter nur ihr Trägerobjekt von der Klasse `KUNDE` und von diesem Objekt benutzt sie nur die eine Anfrage `adresse`. Man kann hier also das Parameterobjekt durch die Anfrage ersetzen und `{KUNDE}.adresse_gueltig` wird zu `ADRESSE.gueltig`.

Hier hat es sich wirklich gelohnt, das Trägerobjekt einer Routine auch als Parameter zu betrachten, den man durch Werte von Anfragen ersetzen kann (oder umgekehrt). Andernfalls hätten wir hier eine neue Refaktorisierungsregel erfinden müssen “Routine zu Attribut verschieben”, und dann käme auch noch “Routine zu Attribut von Parameter verschieben” und so weiter. Aber durch die gewählte Allgemeinheit brauchen wir nur zwei Refaktorisierungen, um daraus alle anderen zusammzusetzen. Wir brauchen “Parameter zu Träger rotieren” (die Routine wird so verschoben, dass ein Parameter zum Trägerobjekt wird) und “Objekt durch Anfrage(n)werte ersetzen” (und umgekehrt).

Fowler behandelt das Verschieben von Feldern ähnlich wie das Verschieben von Routinen und geht nicht darauf ein, dass es sich dabei auch um eine Änderung des Datenmodells handelt, die weit reichende Auswirkungen haben kann. In der Regel verwendet man diese Verschiebung nur zwischen Klassen, die in einer Eins-zu-Eins-Beziehung stehen (oder Vielleicht-zu-Eins), wir werden dafür gleich noch ein Beispiel sehen. Oftmals verschiebt man auch Felder zwischen Klassen, um danach die Kardinalität ihrer Beziehung zu ändern. Zum Beispiel könnte man die Felder `strasse`, `ort` von `KUNDE` in eine neue Klasse `ADRESSE` auslagern und danach mehrere Adressen pro Kunde erlauben.

7.3.4 Unterarten oder Unterklassen?

Wenn es von einer Klasse verschiedene Arten von Instanzen gibt, braucht man nicht immer gleich Unterklassen. Einfacher kann man die Unterarten durch ein Attribut unterscheiden (dass in [7] “Typschlüssel” genannt wird). Ich nenne solche Attribute “Artenschlüssel”, damit man nicht denkt, sie hätten etwas mit dem Typsystem der Programmiersprache zu tun. Hier ein Beispiel (von Bertrand Meyer [31]):

```
class KID feature
  room_mate : KID
end -- class KID
```

```

class BOY redefine room_mate feature
  room_mate : BOY
end -- class BOY

class GIRL redefine room_mate feature
  room_mate : GIRL
end -- class GIRL

```

Mit diesem Code hat Meyer den Versuch gemacht, eine Invariante mit Hilfe des Typsystems auszudrücken, nämlich das Jungs nur mit Jungs ihr Zimmer teilen können, und Mädels nur mit Mädels. In [44] erkläre ich, warum das Typsystem dafür schlecht geeignet ist. Eine einfachere Lösung für das Problem verwendet einen Bool'schen Artenschlüssel:

```

class KID feature
  is_boy : BOOL
  room_mate : KID
invariant
  room_mate.is_boy = is_boy
end -- class KID

```

Artenschlüssel sind einfacher zu implementieren als Unterklassen, und erst wenn man mehrere Fallunterscheidungen anhand der Schlüssel macht, wird man wirklich Unterklassen verwenden. Dies ist eine typische Abwägung zwischen einer einfachen Lösung, mit der man meist anfängt, und der flexibleren Variante, zu der man faktoriert, wenn das Programm komplexer wird.

Das Prinzip der Einzigsten Auswahl

Bertrand Meyer postuliert in seinem Jahrhundertwerk [29] das Prinzip der Einzigsten Auswahl (*Single Choice Principle*), welches grob gesagt immer die dynamische Bindung gegenüber Fallunterscheidungen bevorzugt, und damit Untertypen gegenüber Unterarten. Viele Kritiker meinen, dass sich beide Varianten nur darin unterscheiden, welche Teile des Codes zusammen gespeichert werden und welche nicht. Im Beispiel mit Unterklassen hätten wir dann alle Routinen der Jungs in einer Klasse und alle Routinen für Mädels in einer anderen; mit Unterarten wären jeweils der Code für Jungs und für Mädels zusammen in je einer Routine. Die Kritiker meinen dann weiter, dass folglich Unterklassen nützlich sind, wenn man öfter mal neue Arten hinzu fügt (man braucht dann nur eine neue Klasse, anstatt in jeder Routine einen Fall hinzuzufügen), und dass folglich Unterarten nützlich sind, wenn man öfter mal neue Routinen hinzu fügt (man braucht dann nur eine neue Routine mit allen Fällen, an Stelle eine neuer Routine in jeder Klasse).

Diese Überlegungen gehen aber völlig an der Sache vorbei! Erstens haben wir ja schon gelernt, dass wir Code nicht erweiterbarer als so einfach wie möglich schreiben können, und dass Spekulationen auf mögliche Erweiterungen zur Einfachheit ganz bestimmt nicht beitragen! Wenn eine Software-Erweiterung so einfach wäre, wie nur ein paar neue Routinen oder ein paar neue Klassen, dann wäre es auch ziemlich egal, welche der beiden Lösungen wir nun wählen. In jedem Fall muss man sicherstellen, dass jede Routine alle Fälle abdeckt bzw. jede Klasse alle Routinen implementiert. Und diese Arbeit ist so oder so nicht leichter.

XP'ler meinen ja, Unterklassen seien besser, weil dann der Compiler prüfen kann, ob jede Routine in jeder Klasse implementiert wird. Gleichzeitig empfehlen die XP'ler aber auch, dass man für jede Routine eine Standard-Implementierung in der Oberklasse angibt, und dann kann der Compiler schon nicht mehr helfen. Er weiß ja nicht, ob die Routine absichtlich nicht redefiniert wurde.

Damit wissen wir aber immer noch nicht, worum es bei der Einzigsten Auswahl wirklich geht. Und der normale Durchschnittsprogrammierer kann darauf gar nicht kommen, weil ihm das entscheidende Denkwerkzeug dazu fehlt: Klasseninvarianten. In den einfachen Fällen von Jungs

und Mädels ist es wirklich egal, ob man Unterarten oder Unterklassen implementiert. Aber bei wirklich nicht-trivialen Systemen haben Klassen ein Implementierungsgeheimnis und geheime Attribute, die mit Hilfe der Klasseninvariante vor Chaos geschützt werden. Würde man so ein System nur mit Artenschlüsseln implementieren, so müsste jede Routine die Geheimnisse aller Klassen kennen — die Geheimnisse wären in den Routinen vermischt. Aber in der Lösung mit Vererbung sind die Gemeinsamkeiten aller Routinen als Spezifikation oder Implementierung in der Oberklasse festgelegt und die Unterschiede befinden sich in den einzelnen Klassen.

Wieder einmal sehen wir, dass die Strukturierungsmöglichkeiten der modernen Programmiersprachen nur leere Hüllen sind, die durch Abstraktion in Form von Geheimnissen erst gefüllt werden.

Ein fast schon übertrieben vorbildliches Beispiel für die Invarianten sind übrigens die Implementierungen von abstrakten Datenstrukturen (Mengen, Sequenzen, Tabellen, ...). Jede Implementierung bietet die selben Routinen an, aber auf ganz unterschiedlicher interner Grundlage. (In jedem Fall eine Mischung aus Arrays und Referenzen, die erst durch die Invariante ihre Identität erhält. Siehe auch 6.5.)

7.3.5 Werte, Objekte oder Zustände?

Wir haben im letzten Kapitel schon den Unterschied zwischen algebraischen Objekten und imperativen Objekten gesehen. Letztere sind ja genau dann mit besonderer Vorsicht zu behandeln, wenn man innerhalb eines Programms mehrere Referenzen auf so ein veränderliches Objekt hat. Deswegen nennt man sie auch Referenzobjekte und ihre einfacheren Kollegen nennt man Wertobjekte. Wir haben über Referenzen auch schon gelernt, dass alle Referenzobjekte zusammen den *Zustand* eines objektorientierten Systems ausmachen. Ein Entwickler muss zu jedem Zeitpunkt der Programmausführung wissen, wie viele Objekte von jeder Referenzklasse allokiert sind und wie diese verbunden sind. Wenn man ein Objekt dieser Struktur ändert, so ändert man die komplette Struktur.

Die Kommunikation zwischen dem Computer und der Außenwelt findet immer über Werte statt (Zahlen und Zeichenketten), und diese müssen erst in Objekte übersetzt werden. In der Regel verwendet man dazu Datenstrukturen, die auf die Zustandsobjekte verweisen. (Für Wertobjekte kann man einfache Konstruktoren verwenden.) Oftmals muss man sich beim Entwurf entscheiden, ob ein bestimmter Wert im Programm durch einen einfachen Wert (Zeichenkette, Zahl) repräsentiert werden soll oder durch ein Objekt einer eigens erstellten Klasse. Im folgenden Groß-Beispiel über Diplomacy (siehe Abschnitt 3.6) würde es fast ausreichen, jedem Raum (Objekt der Klasse (SPACE) eine Liste seiner Küsten als Zeichenkette zuzuordnen. Aber dann stellt man fest, dass Paare von Raum und Küste oft verwendet werden, und man schließt sie zu einer eigenen Klasse namens Gebiet (AREA) zusammen.

Wenn man mit einfachen Werten arbeitet, verwendet man eher *explizite*, ganzheitliche Datenstrukturen wie Mengen und Abbildungen, und bei der Arbeit mit Objekten modelliert man Relationen eher *implizit*, Punkt-bezogen: Referenzen zwischen Objekten und Bool'schen Attributen für Mengen-Zugehörigkeit. Dieser Unterschied tritt sehr schön zu Tage, wenn man den folgenden Code mit der formalen Spezifikation in B [49] vergleicht. Hier ein Ausschnitt aus der Wurzel-Klasse des Beispiels:

```
class DIPLOMACY_MAP
creation make
feature
  immutable spaces : BY_MAP[STRING, SPACE]
  immutable powers : BY_MAP[STRING, POWER]
feature {NONE}
  make is
  do
    create spaces.make(agent {SPACE}.name)
    create spaces.make(agent {POWER}.name)
```

```

    ...
end
end -- class DIPLOMACY_MAP

```

Die beiden Datenstrukturen `spaces` und `powers` sind die Haupteintrittspunkte in das Netzwerk von Objekten, das die Kartenstruktur und den Spielzustand speichert. Dabei wird eine spezielle neue Datenstruktur verwendet, die ich kürzlich erfunden habe (als Teil von DESSY [41]). Es handelt sich dabei um eine Abbildung (*map*) von einem Typ A zu einem Typ B, wobei die Schlüssel *a* jeweils ein Attribut des gespeicherten Objekts *b* sind. Im Beispiel wird jeder Raum (`SPACE`) und jede Macht (`POWER`) durch seinen Namen gespeichert. By-Maps haben den Vorteil, dass man den Schlüssel nicht redundant in der Klasse und der Datenstruktur speichern muss, und dass sie eine einfachere Schnittstelle erhalten. Das Schlüssel-Attribut wird bei der Konstruktion einer By-Map als Funktion übergeben (und kann dann natürlich nicht mehr geändert werden). Ich habe den Konstruktor-Code hinzugefügt, weil diese Information für das Datenmodell wichtig ist.

Im folgenden Code drückt sich eine wichtige Entwurfsentscheidung nur indirekt aus, und daher nenne ich sie hier explizit. Bei der Diplomacy-Entscheidung geht man von einer festen Kartenstruktur aus (mit den Provinzen, Meeren und ihren Nachbarschaften), auf denen sich die Einheiten variabel bewegen. Im Programm wird dies dadurch ausgedrückt, dass die Kartenobjekte bzw. die Karten-spezifischen Attribute unveränderlich sind. Die Daten zum Spielzustand hingegen werden als veränderliche Attribute gespeichert. Die Kartenstruktur wird nur einmal geladen, wenn die Objekte erstellt werden; man erkennt Struktur-Attribute also daran, dass sie als `immutable` deklariert sind. Objekte der Klassen `SPACE`, `AREA`, `POWER` werden auch nur beim Laden der Karte erstellt, und während der Spielzüge bleiben die Objekt-Mengen gleich, nur die Relationen und Wert-Attribute ändern sich. (Hier sieht man schon, dass man das Klassenmodell nur vernünftig planen kann, wenn man die Funktionalität in Betracht zieht. Wenn man die Objekt-Erstellung nicht von Anfang an im Entwurf betrachtet, muss man sonst gleich am Anfang der Implementierung den Entwurf wieder umbiegen. (Vielleicht machen die XP'ler ja keinen a priori Entwurf, weil sie damit überfordert sind.)

Weiterhin gilt für den folgenden Code, dass alle Attribute non-Void sein sollen, es sei denn dies ist explizit gekennzeichnet. Der Kommentar "maybe" sagt, dass für ein Attribut auch `Void` ein sinnvoller Wert sein kann. Zum Beispiel `SPACE.occupier = Void` für ein unbesetztes Feld. Hier die Klassenschnittstellen von `POWER` und `UNIT`:

```

class POWER interface feature
  immutable name : STRING
  immutable home_provinces : SET[SPACE]
  immutable army : UNIT
    ensure Result.is_army and Result.power = Current
  immutable fleet : UNIT
    ensure Result.is_fleet and Result.power = Current

  ressource_centers : SET[SPACE]
  number_of_units : INTEGER
end -- class interface POWER

immutable feature
  is_army : BOOLEAN
  is_fleet : BOOLEAN
  owner : POWER
invariant
  is_army /= is_fleet
end -- class interface UNIT

```

Das `/=` (Ungleich) in der Invariante liest man als "entweder .. oder".

Die Einheiten werden durch die einfache Werteklasse `UNIT` dargestellt; man beachte wie elegant der Einheitentyp hinter Bool'schen Anfragen gekapselt ist. Ein expliziter Artenschlüssel wäre hier eine unnötige Indirektion, und er wäre auch länger zu schreiben: `u.type = u.Army.type` an Stelle des simplen `u.is_army`. Ersteres ähnelt doch zu sehr dem Anfängerfehler `x.is_empty = True` an Stelle von `x.is_empty`. Die Platzierung der Fabrikroutinen für Einheiten in der Klasse `POWER` mag zwar zunächst überraschend erscheinen, aber dort lassen sie sich am einfachsten verwenden (sei `p` eine Macht, dann ist `p.army` eine Armee dieser Macht) und sogar einfach implementieren: man schreibe in den Konstruktor von Klasse `POWER`...

```
create army.make(Current, True)
create fleet.make(Current, False)
```

(Die Anfragen `army` und `fleet` werden also als Attribute implementiert!) Die Klasse `UNIT` implementiert man dann mit zwei Attributen, einer Anfrage und dem trivialen Konstruktor, der auch nur von `POWER` verwendet wird. Selbst bei so einer kleiner Record-ähnlichen Klasse profitieren wir also schon von der Abstraktion! Und als Nebeneffekt erhalten wir noch eine gemeinsame Speichernutzung der `UNIT`-Objekte.

Hier die Schnittstellen der Klassen `SPACE` und `AREA`:

```
class interface SPACE feature
  immutable is_sea, is_province, is_inland, is_coastal : BOOLEAN
  immutable has_rc : BOOLEAN
  immutable areas : BY_MAP[STRING, AREA] -- by 'coast'
  immutable neighbour_spaces : SET[SPACE]

  occupier : UNIT -- maybe
  occupied_coast : STRING -- maybe
  rc_owner : POWER -- maybe
invariant
  is_sea /= is_province
  is_province implies (is_inland /= is_coastal)
  (is_inland or is_coastal) implies is_province
-- consequences: 'is_sea implies not is_inland' etc.

  is_rc implies is_province
  not is_rc implies rc_owner = Void

  (is_sea or is_inland) implies areas.is_empty
  areas.for_all((a : AREA)
    do Result := a.space = Current end
  )
neighbour_symmetric:
  neighbour_spaces.for_all((s : SPACE)
    do Result := s.neighbour_spaces.has(Current) end
  )
end -- class interface SPACE

immutable class interface AREA feature
  space : SPACE
  coast : STRING
  neighbour_areas : SET[AREA]
  neighbour_spaces : SET[SPACE]
invariant
  space.areas.has(Current)
```

```

neighbour_symmetric:
  neighbour_areas.for_all((a : AREA)
    do Result := a.neighbour_areas.has(Current) end
  )
neighbour_superset:
  neighbour_spaces.for_all((s : SPACE)
    do Result := s.neighbour_spaces.has(space) end
  )
end -- class interface AREA

```

Die seltsame Syntax in den Invarianten steht für anonyme Funktionen, die als Parameter übergeben werden. Der letzte Aufruf ist zum Beispiel äquivalent zu folgendem HASKELL-Code: `neighbour_spaces 'for_all' (λ s → (neighbour_spaces s) 'has' space)`.

Für die Artenschlüssel werden hier wieder die eleganten Bool'schen Anfragen benutzt. Man kann diese leicht durch zwei Attribute und zwei daraus abgeleitete Anfragen implementieren.

Man beachte auch, wie wir mit den Attributen `areas` und `space` die Eins-zu-N Beziehung zwischen `SPACE` und `AREA` auf bidirektionale Weise implementiert haben. Eine solche Navigierbarkeit in beide Richtungen ist eine Redundanz, die man zugunsten der Effizienz einführt — wir verlassen hier also eindeutig den Bereich "Programmieren als Spezifizieren". Dementsprechend ist die Invariante auch keine Beschreibung der Anwendungswelt mehr (bei den Artenschlüsseln war das ja noch der Fall), sondern sie gibt eine Konsistenz-Bedingung der Implementierung an. Wenn man die Invarianten beider Klassen zusammen betrachtet, erhält man übrigens die Aussage, dass die Mengen `areas` aller möglichen `SPACE` Objekte disjunkt sind — eine solche Objekt-übergreifende Aussage kann man direkt gar nicht aufschreiben.

Und hier die Schnittstelle der letzten Klasse aus dem Beispiel:

```

immutable class interface ORDER feature
  orgin, target : SPACE
  target_area : AREA -- maybe

  is_hold, is_move, is_support, is_convoy : BOOLEAN
  is_convoyed : BOOLEAN
  is_direct : BOOLEAN is -- ensure
    do Result := is_move and not is_convoyed end

  supported : ORDER -- maybe
invariant
  target_area /= Void implies target_area.space = target

  is_convoyed implies is_move

  (is_support or is_convoy) = (supported /= Void)
end -- class interface ORDER

```

Und ein weiteres Mal kommen hier die Bool'schen Anfragen zum Tragen, allerdings ist hier die Invariante nicht voll formal ausgeschrieben: immer genau eines `is_hold`, `is_move`, `is_support`, `is_convoy` trifft zu, dies sieht aber formalisiert nicht sehr elegant aus. Auch für die Implementierung dieser vier Zustände scheint zunächst ein Ganzzahl-Attribut angebracht zu sein als mehrere Bool'sche Attribute, aber nach nochmaligem Hinschauen erkennen wir, dass die "Voidness" von `supported` ja benutzt werden kann, also brauchen wir neben `is_convoyed` nur noch ein weiteres Bool'sches Attribut und die vier Anfragen können dann als Funktionen implementiert werden.

Man mag sich fragen, warum es bei all diesen verschiedenen Anfragen und Typen nicht vielleicht doch angebracht ist, Unterklassen und Vererbung zu verwenden. Hier muss ich zugeben,

dass ich den Entwurf unter Kenntnis der Programmfunktionalität gemacht habe, insbesondere der Erfahrung aus der formalen Spezifikation. Eigentlich haben wir ja gelernt, Anfragen wie `if a.hat_diesen_Typ then ... elseif a.hat_jenen_Typ then ...` zu vermeiden, aber oftmals und auch in diesem Beispiel spielen die Typen verschiedener Objekte eine Rolle und die Konsequenzen der Fallunterscheidung lassen auch schlecht den entsprechenden Klassen zuordnen. Im unserem Fall haben wir zum Beispiel die Anfrage:

```
-- in class UNIT
can_occupy( s : SPACE ) : BOOLEAN is
do -- ensure
    Result := ( is_fleet and (s.is_coastal or s.is_sea) )
              or ( is_army and s.is_province )
end -- can_occupy
```

Hätten wir Unterklassen für die Arten von Einheiten, so würde zum Beispiel in der Klasse `ARMY` nur noch stehen: `Result := is_province`. Natürlich wäre das eine Art von Vereinfachung, aber bei einer so kurzen Routine wie `tt can_occupy` ist es sicher besser, alle Fälle auf einen Blick zu sehen als auf verschiedene Klassen verstreut.

Aus diesem Beispiel können wir zwei wichtige Lehren ziehen: erstens sieht man hier, dass der Entwurf wesentlich leichter und zielgerichteter vonstatten geht, wenn man sich auf die Schnittstellen und Invarianten konzentriert. Die Bool'schen Anfragen sind für die Kunden der entworfenen Klassen absolut einfach zu verwenden, und ihre Implementierung ist komplett versteckt, also brauchen wir uns beim Entwurf auch nicht übermäßig darum zu kümmern.

Die zweite allgemeine Feststellung bezieht sich auf die Entscheidung Kartenstruktur und Spielzustand in den selben Klassen zu speichern. Ganz sicher macht das die Programmierung einfacher, man kann sich jetzt fragen, welche Flexibilität man dadurch einbüßt. Die Antwort: mit diesem Entwurf ist es nicht möglich, verschiedene Zustände des Spielverlaufs (oder unterschiedlicher Spiele) zu repräsentieren. (Zumindest nicht, ohne die Kartenstruktur auch mehrfach zu speichern.) Aber eine Alternative ist ganz einfach zu erstellen: man trennt die Klassen `SPACE` und `POWER` einfach auf in einen Teil Spielzustand und einen Teil Kartenstruktur und erstellt eine Klasse `DIPLOMACY_STATE`, die die Zustandsobjekte beisammen hält, so wie es `DIPLOMACY_MAP` mit der Kartenstruktur tut. Die Objekte des Spielzustandes erhalten dabei einen Link auf die der Kartenstruktur, in der Gegenrichtung braucht man keine Verknüpfung, da Zugriffe immer von einer Spielsituation ausgehen. (Alternativ könnte man die Links auch weglassen und Zugriffe immer über die Namen und die Tabellen in `DIPLOMACY_MAP` implementieren.)

Die Objekte der Kartenstruktur werden dann komplett unveränderlich. In diesem Entwurf kann man dann ganz einfach für jeden Spielzustand ein neues Objekt der Klasse `DIPLOMACY_STATE` mit seinen Unterobjekten hinzufügen. (Dabei könnte sich auch herausstellen, dass auch die Zustands-Objekte eigentlich unveränderlich sein sollten, zum Beispiel weil man bei jedem Spielzug sowieso den alten Spielzustand aufhebt. Auch diese Refaktorisierung kann man nachträglich noch machen.)

Wir sehen also, dass man in einem guten objektorientierten Entwurf durchaus einzelne Aspekte behandeln kann, ohne spezielle Sprach-Erweiterungen zum Aspekt-orientierten Programmieren zu benutzen.

7.3.6 Vererbung oder Delegation?

Im Abschnitt über Vererbung habe ich bereits ausführlich die Unterschiede zwischen Vererbung und Kunde-Anwender-Beziehung besprochen. In vielen Fällen ist völlig klar, welchen Mechanismus man braucht: Polymorphismus geht nur über Kunde-Anbieter, Wiederverwendung von Spezifikationen nur über Vererbung, ...

Wenn ein Kunde viele der Routinen eines Anbieters selbst wieder anbietet (über eigene Stummel-Routinen, die nichts tun als den Anbieter aufzurufen), so nennt man das *Delegation*.

Mit Vererbung würde man den selben Effekt ohne die Stummel-Routinen erreichen. Andererseits, wenn man viel von einer Klasse erbt und nicht deren ganze Schnittstelle reexportiert bzw. nicht besonders viele ihrer Routinen überschreibt, so könnte es sein, dass eine Kunde-Anbieter-Beziehung die beiden Klassen besser verbindet: so erhält der Anbieter eine unabhängige Schnittstelle und kann auch von anderen Klassen einfacher genutzt werden.

Zum Beispiel kann man alle Klassen, die eine Post-Adresse besitzen zu Erben einer Klasse ADRESSABLE machen, die entsprechende Funktionalität enthält. Alternativ kann man all diese Klassen zu Kunden einer Klasse ADRESS machen. In diesem Beispiel scheint die zweite Lösung intuitiv passender zu sein. In der Tat gibt es bei Fowler (und in einschlägigen Tools) auch nur ein Refaktorisieren weg von der einfacheren Vererbung hin zur flexibleren Delegation.

7.3.7 Singleton oder Multiton?

Wie wir im vorigen Kapitel gesehen haben, ist es eine der wesentlichen Grundideen der Objekt-Orientierung, dass man ein Modul in Form einer Klasse nur genau einmal programmiert und dann beliebig viele Instanzen davon verwenden kann. In manchen Fällen braucht man aber pro Programm nur eine Instanz (das ist einfach, man erstellt eben nur eine), und diese soll von vielen anderen Objekten gemeinsam genutzt werden. Das nennt man dann ein Singleton-Objekt (einer Singleton-Klasse), und man implementiert es in der Regel über eine globale Referenz-Konstante.

Stünde diese Möglichkeit nicht zur Verfügung, so müsste man Referenzen auf dieses Singleton-Objekt ständig in Aufrufen übergeben und in Objekten zwischenspeichern. Das Singleton-Muster erleichtert das Programmieren daher erheblich. Andererseits verliert man mit der Bindung an das Singleton-Muster aber auch einige Flexibilität.

In einem Programm zur Verwaltung eines Frachters mag das Objekt FRACHTER ein Singleton sein, das von vielen anderen Objekten (DECK, LADUNG, MANN-SCHAFT, ..) referenziert wird. Das macht die Programmierung einfach. Aber wenn man das Programm zur Verwaltung einer Flotte erweitern will, so genügt es nicht mehr einfach den Frachter mehrfach zu instantiiieren: man muss auch allen Objekten eine explizite Referenz auf ihren Frachter geben und diese korrekt initialisieren.

Während in der Literatur meist nur erklärt wird, wie man Singletons implementiert, sollte ein Programmierer also stets die bewusste Entscheidung treffen, was er als Singleton und was als normales Objekt implementiert.

7.3.8 Refaktorisieren abstrakt

Die kleinschrittige Refaktorisierungsanleitung von Fowler geht immer nach dem gleichen Muster vor. Ich fragte mich dabei, ob man sich nicht diese ganze Lektüre sparen kann, wenn man sich einmal dieses abstrakte Muster bewusst macht; es ist nämlich ganz einfach:

1. Spezifiziere das neue Element und erstelle Testfälle dafür.

Das neue Element kann eine ausfaktorierte Routine sein, eine neue Klasse oder die neue Implementierung einer verschobenen Routine. Außerdem kompiliert man schon einmal, um Syntax- und Typfehler auszuschließen.

2. Implementiere das neue Element, normalerweise durch Kopieren und Anpassen von Code. Dann lässt man gleich den schon vorbereiteten Test laufen.

Das Kopieren ist dabei nur zeitweilig, damit man während dem Refaktorisieren den alten Code noch behalten kann. (Und die Anpassungen sind oft automatisierbar, aber das ist eine andere Geschichte.) Wenn man ganz vorsichtig ist oder die alte Routine spezifikativ

war (d.h. keine prüfbare Spezifikation hatte), kann man auch zusätzlich beide Versionen im Test vergleichen.

Dadurch haben wir bis jetzt noch keinen Buchstaben am Programm geändert (sondern nur hinzugefügt). Wir könnten also theoretisch beruhigt Feierabend machen und später weiter arbeiten, das Programm läuft trotzdem noch wie zuvor. Wenn der Test erfolgreich war, haben wir schon den wichtigsten Schritt geschafft und das ohne Risiko für das Programm!

3. Nun Implementieren wir die alte Version des Elements durch eine einfache Delegation auf die neue Implementierung.

Damit wird die durch das Kopieren im vorigen Schritt entstandene Redundanz wieder entfernt. Diesen Test würde man jetzt wieder entfernen.

Dadurch haben wir bis hierher schon die Klasse refaktoriert, ohne die alte Schnittstelle zu verändern. Obwohl der neue (und bereits getestete) Code jetzt schon vom Programm verwendet wird, mussten wir noch keinen anderen Programmteil ändern.

4. Jetzt können wir jeden Kunden der Schnittstelle (einzeln) anpassen, so dass er die neue Schnittstelle verwendet.

Dies ist besonders einfach, weil man im Prinzip nur ein Inlining des alten Elementes durchführen muss: einen Aufruf durch seine (triviale) Implementierung ersetzen.

Weil wir uns jeden Kunden einzeln vornehmen (und natürlich dazwischen testen), können wir wieder jederzeit zwischendurch nach Hause gehen. Dies ist ganz besonders wichtig, weil wir vielleicht gar nicht alle Kunden der Schnittstelle selbst geschrieben haben und ändern können. Dann müssen wir allen Kunden-Programmierern Zeit lassen, auf die neue Schnittstelle "umzusteigen". Wenn die Schnittstelle Welt-öffentlich ist (d.h. es gibt Kunden-Programmierer auch in anderen, unabhängigen Firmen), kann dieser Umstieg natürlich sehr lange dauern.

Möglicherweise wollen wir aber auch die alte Schnittstelle oder einen Teil davon beibehalten. Sei es, weil die Delegation gar nicht so trivial ist, oder weil wir die neue Version nur intern verwenden wollen. Typisches Beispiel ist das Ausfaktorieren einer neuen Routine: ob die neue nun geheim oder veröffentlicht ist — die alte wird behalten. Dann müssen wir die Kunden gar nicht (alle) ändern und sind jetzt schon fertig.

5. Falls die alte Version nicht mehr benötigt wird, können wir sie nun endlich löschen; zusammen mit ihrem Test-Code und anderen Elementen, die nur von ihr noch verwendet wurden.

Erst im letzten Schritt kommen wir also zu der eigentlichen Netto-Vereinfachung. Es ist natürlich psychologisch für den Programmierer schwierig, so lange zu arbeiten und erst so spät das Resultat zu sehen. (Bei Welt-öffentlichen Schnittstellen möglicherweise mehrere Jahre!) Deswegen sind Programmierer auch oft versucht, die Änderung ganz in einem Schritt vorzunehmen, wobei man wegen dem stupiden Kopieren-und-Anpassen oft Leichtsinnsfehler macht und sich verzettelt. Und dann bevorzugt man es lieber, den Code so zu lassen wie er ist.

Genauso wie die Arbeit mit Zusicherungen ist das Refaktoriieren eben eine Technik für die man erst einmal ein bisschen investieren muss, bevor sie sich vielfach auszahlt. Aber in der schnell-lebigen Welt der Informatik, wo so viele Techniken auf den Markt stürzen, die sich nicht rentieren, sind die Menschen misstrauisch geworden: man akzeptiert nur noch Dinge, die sich ohne Umstiegskosten möglichst sofort auszahlen. (Daher zum Beispiel auch der Erfolg aller auf C basierten Sprachen!)

Ich hoffe nun sehr, dass mit wachsender Werkzeug-Unterstützung das Refaktoriieren seinen stupiden Anteil verliert und Ergebnisse schneller sichtbar werden. Eine besondere Herausforderung bleibt dabei die Evolution von Welt-öffentlichen Schnittstellen. Kann man Schnittstellen-Kunden selbst in anderen Institutionen (teilweise) automatisch refaktoriieren? In einem Zeitalter, wo jedes Programm unzählige Komponenten anderer Hersteller benutzt

(man denke allein an die vielen Bestandteile des Betriebssystems und der Laufzeitumgebung der Programmiersprache!), ist die Evolution von inter-institutionellen Schnittstellen eine Voraussetzung, um Programme wirklich immer einfach zu halten und von den Altlasten der Informatik-Geschichte zu befreien. Fünfzig Prozent des Microsoft Windows Quellcodes dienen der Kompatibilität mit alten Versionen — oder noch mehr?

Zum Abschluss sei ein Merksatz wiederholt:

Die Kenntnis des Refaktorisierens ist nicht nur wichtig, um den Entwurf vorhandener Software zu verbessern, sondern noch viel mehr, um schon beim Entwerfen die Alternativen zu möglichen Lösungs-Konstrukten zu erkennen, und sich bewusst zwischen diesen zu entscheiden.

Wir haben ja schon im Kapitel über Qualität gelernt: man sollte versuchen, von Anfang an die Dinge richtig zu machen, denn korrigieren muss man danach trotzdem noch. Wenn man sich aber denkt “ich kann ja später noch refaktorisieren”, dann wird man es nie weit bringen.

8 Und weiter?

“Changing the size of the body of a LaTeX document’s text is a surprisingly difficult task: the best advice to the beginner is ‘don’t do it.’”

— aus einem LaTeX-Ratgeber im Internet.

Als ich vor vielen Jahren mit den Recherchen anfang, aus denen diese Arbeit hervorgeht, war es mein Ziel eine Schritt-für-Schritt-Anleitung für das Programmieren zu geben: Welche Dinge muss man mindestens wissen, welche Dinge muss man mindestens können? Oder anders gesagt: Welche Bücher muss man mindestens gelesen haben und welche Übungen gemacht? Ich würde gerne zu einem Anfänger sagen können: wenn Du dies und jenes kannst, dann kannst Du produktiv und professionell als Programmierer arbeiten und dabei Deine Fähigkeiten weiter entwickeln.

Einen solchen Wissenskanon konnte ich aber nicht zusammenstellen. Stattdessen habe ich nur ein paar Dinge ausmachen können, die mit großer Wahrscheinlichkeit dazu gehören. (Darunter alle der zitierten Bücher und Artikel, es sei denn sie sind explizit als schlechtes Beispiel zitiert.) Aber zwischen diesen Wissens-Inseln klaffen noch zu große Lücken. Die bekannten Texte lassen noch zu viel Interpretationspielraum. Nicht anders ist zu erklären, dass selbst Informatik-Absolventen oftmals fürchterlich schlecht programmieren (obwohl in Sprachen, die sie beherrschen). Derselbe Mangel trifft auch auf die vorliegende Arbeit zu: sie erwartet noch zu viel von ihrem Leser. Auch die zitierten Werke kann man nicht einfach so hinnehmen, allein schon weil sie oft von ganz widersprüchlichen Sichtweisen ausgehen.

Bei Computeranwendungen trifft man überhaupt viel zu häufig auf Dokumentationen wie “Benutzen Sie diese Option nur, wenn sie wissen, was sie tun.” In diesem Sinn ist auch das am Abschnittsanfang gegebene Zitat zu verstehen. Angesichts solcher Schwierigkeiten bleibt Programmierern oftmals nichts anderes übrig, als einfach Lösungen bei anderen abzuschauen. Dass man neuerdings haufenweise Quellcode im Internet findet, hat natürlich diesen Trend noch verschärft. Und für den Anfänger mag diese Taktik auch sehr Gewinn bringend erscheinen: was vorhin noch wie ein unbewältigbar komplexes Problem aussah, wird im Nu von einer noch komplexeren Lösung aus dem Internet schon fast zufrieden stellend gelöst. Dabei hat aber dieses Vorgehen ganz unakzeptable Schwachstellen, insbesondere weil es völlig unberechenbar ist: man kann weder Aufwand noch Dauer einer Lösung einschätzen, weil man nie weiß zu welchem Unterproblem es schon auffindbare Lösungen gibt. Der Schritt von “fast zufrieden stellend” bis zum Ende wird dann absolut unverhältnismäßig viel komplizierter als der ganze große Teil bis dahin.

Für mich ist dieses “Programmieren per Download” (oder “Google-and-Paste”) nur ein Symptom für die Unreife des ganzen Feldes. Schließlich ist die Ursache dieser Probleme ja auch oft nur, dass die gefundenen Programme schlecht programmiert sind und schon gar nicht ordentlich dokumentiert.

8.1 Was ich erreicht habe

The Babylon Project was our last, best hope for peace. It failed. But in the year of the Shadow War, it became something greater: our last, best hope — for victory.
The year is 2260. The place: Babylon 5.

Das Ziel haben wir also noch nicht erreicht. Aber was haben wir stattdessen? Ich denke, dass meine Arbeit in folgenden Dingen dem Stand der Technik voraus ist (Stichpunkte in Reihenfolge der Kapitel):

- “Programme sind (auch) Spezifikationen” ist eine oft publizierte Aussage der Verfeinerungstheorie. Hier wird sie jedoch zum ersten Mal auf das praktische Programmieren angewendet (u.a. mit dem `do -- ensure` Konstrukt) und sogar mit dem Extremen Programmieren in Verbindung gebracht.
- “Gute und schlechte Redundanz” wurde schon 1978 von Bertrand Meyer und Claude Baudoin erwähnt, aber nur in einem Absatz des letzten Kapitels ihres Buches. Ich habe das Konzept wieder ausgegraben und stelle es an den Anfang meiner Arbeit, wo es hingehört.
- Dass sich die äußeren Qualitätsmerkmale (außer der Effizienz) nur auf den spezifikativen Teil der Programmierung beziehen, ist eine ganz wesentliche Feststellung meiner Methode. Woanders habe ich davon noch nie gelesen.
- Die Arbeiten von Michael Jackson zu Begriffsbestimmungen haben sich noch nicht weit durchgesetzt. Mein Beitrag sie zu erklären geht über die reine Übersetzung hinaus, insbesondere durch den Begriff der “externen Begriffe” (Begriffsdopplung inbegriffen).
- Für die Lehre und Forschung der Programmierung sind nicht-triviale und doch überschaubare Fallstudien unerlässlich. Mit der “Diplomacy-Bewertung” habe ich der bekannten Menge ein ergiebige Beispiel hinzugefügt.

Dass meine informale Spezifikation des Bewertungsalgorithmus wesentlich prägnanter ist als die vorher publizierten, ist dabei für den allgemeinen Fortschritt der Informatik wohl nicht so bahnbrechend. Immerhin verdeutlicht es die Bedeutung von zahlreich verwendeten Definitionen — einer Heuristik, die ich mich auch nicht erinnern kann, woanders schon einmal gelesen zu haben.

- Mein Abriss über die formalen Theorien der Programmierung enthält wirklich nur schon allgemein anerkannte Ideen (abgesehen vielleicht von APTOP), allerdings halte ich meine Präsentation für recht innovativ, weil sie den Fortschritt der Theorien im Laufe der Entwicklung und den Gegensatz zwischen “philosophischen und praktischen” Theorien zusammen mit ihrer schließlichen Versöhnung so schön hervorhebt.
- Spezifikationen und Tests werden oft miteinander verglichen oder ineinander umgewandelt, dass aber Tests auch nur eine Art von prüfenden Spezifikationen sind und dass Spezifikationen zusammen mit Eingabe-Generatoren auch gleichwertig mit Tests sind — davon habe ich noch nichts gelesen.
- Die Stufen der “Operationalisierungshierarchie” habe ich mir selbst ausgedacht.
- Kapitel 6 bietet einen Haufen bekannten Materials in neuer Darstellung. Ganz wichtig und neu ist die Idee, in objektorientierten Sprachen funktional zu denken, und zwar ganz ohne spezielle Konstrukte der Sprachen zu verwenden oder sie sogar zu erweitern.
- Spezifikationen und Verträge gleichberechtigt jeweils zusammen mit den implementativen Teilen der Sprache vorzustellen ist eine fast zwingende Idee — in der Literatur aber bisher ungesehen, von ganz formalen Werken abgesehen.
- Vererbung — ein kontroverses Thema. Mein Beitrag baut auf Bertrand Meyers Sichten auf und verwendet dabei einige Muster, von denen bisher noch nichts zu lesen war.
- In der Software-Architektur ist noch lange nicht klar, was genau unter Komponenten und Konnektoren zu verstehen ist, insbesondere wenn man sie im Zusammenhang mit dem Entwurf von Spezifikationen und fertigen Programmen sieht. Abschnitt 7.1 bietet dazu eine neue einheitliche Sicht.
- Die komplette Theorie der Alternativen ist neu für den Softwareentwurf.

- Refactoring nach Fowler ist das schrittweise Verändern des Entwurfs existierender Software. Bei mir wird es eingespannt in die Theorie der Alternativen, erhält also eine viel mächtigere Position in einer früheren Phase.

Über die Signifikanz dieser Beiträge erhält man einen Überblick durch Vergleich mit den Thesen, die der Arbeit beiliegen.

8.2 Was man mal erreichen sollte

Zunächst sind da einige wichtige Themen, die in der Arbeit gar nicht behandelt wurden, obwohl sie auch für die Programmierung unerlässlich sind: der Umgang mit Referenzen, Strukturen zur Objekt-Erstellung, Persistenz, und das ausgesprochen weite Feld der Nebenläufigkeit ...

Weiterhin gibt es in dieser Arbeit so wie in der Literatur überhaupt einen chronischen Mangel an Beispielen. Ich meine sogar, dass selbst ein richtig große Fallstudie: ein Programm, mit Spezifikation und Entwurfs-Alternativen dokumentiert, noch als wissenschaftlicher (oder zumindest didaktischer) Fortschritt zu bewerten ist.

Schließlich ergeben sich Forschungsziele aus den beiden Aspekten der Programmierung: beim Formalisieren von Anforderungen zu Programmen braucht man höhere Abstraktionsebenen, Komponenten, Generatoren und auch automatische Refaktorisierungswerkzeuge. Und beim Programmieren aus formalen Spezifikationen sind Verifikationswerkzeuge hilfreich, an denen schon so lange geforscht wird und die es aber noch nicht in die Praxis geschafft haben. Einer der Gründe für dieses Versagen ist sicher auch, dass bisher Programmierer kaum Verträge und andere Zusicherungen verwenden. Wenn man aber erst einmal eine bestimmte Mengen von Zusicherungen im Code hat (und das müssen keine kompletten Spezifikationen sein), dann kann ein Verifikator sehr viele Prüfungen vornehmen und Fehler melden, ohne dass man das Programm ausführen muss. Natürlich kann der Verifikator nicht die vollständige Korrektheit von Programmen beweisen, aber er kann eine Menge Fehler finden, indem er die Konsistenz der redundanten Informationen prüft. Und darauf kommt es ja an. Siehe Kapitel Eins.

8.3 Danksagungen

Es ist schon erstaunlich: diese Arbeit enthält zahlreiche Ideen, die in keiner Literatur-Quelle zu finden sind, aber trotzdem hätte ich ohne die Bücher meine Ideen nie gehabt! Dies gilt für all die von mir geliebten Werke von Hoare, Hehner und Meyer, aber auch für die Gammas, Becks und Shaws, bei deren Lektüre mir so unwohl ist. Ich kann nur benennen feststellen: dies ist also Inspiration.

Einige Menschen haben mich auch ohne den Umweg über Publikationen inspiriert:

- Heidrun Will erzählte mir vom Generieren und Eliminieren von Alternativen. (Deswegen habe ich dazu auch keine Literatur-Referenz.)
- Stefan Schwetschke widersprach mir sehr oft; seine XP-radikale Argumentation führte mich teilweise auf den Schlag zu neuen Ideen, insbesondere beim Thema Verträge-versus-Typen.

Ideen sind inexistent ohne Form. Günter Hübel, Katrin Kunz, Stefan Schwetschke und Heidrun Will haben meine Arbeit trotz großem Zeitdruck ausführlich korrigiert. Katrin strich Kommas heraus und Heidrun fügte sie wieder ein. Die Restfehler sind natürlich alle meine Schuld.

Für die stetige Ermunterung und positive Energie danke ich allen genannten lieben Menschen.

Literaturverzeichnis

- [1] Ralph Back, Jim Grundy, and Joakim von Wright. Structured Computational Proof. Technical Report TUCS-TR-65-25, Turku Centre for Computer Science, 1996. <http://citeseer.ist.psu.edu/article/back96structured.html>.
- [2] R. Bird, T. E. Scruggs, and Mastropieri. *Introduction to Functional Programming*. Prentice Hall, 2nd edition, 1998.
- [3] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. Technical report, 1999. <http://theory.lcs.mit.edu/~edemaine/papers/ResizableArraysTR/>.
- [4] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058, Department of Computer Science, Carnegie-Mellon University, 1977.
- [5] E.W. Dijkstra, D. Parnas, W.H. Scherlis, M.H. van Emden, J. Cohen, R. Hamming, R.M. Karp, and T. Winograd. A debate on teaching computer science. *Communications of the ACM*, 32(12), 1989.
- [6] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia*, volume 19, pages 19–31, Providence RI, 1967. American Mathematical Society.
- [7] Micheal Fowler. *Refactoring — wie Sie das Design vorhandener Software verbessern*. Addison-Wesley, 2000.
- [8] Richard P. Gabriel. *Patterns of Software*. Oxford University Press, 1996.
- [9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
- [10] D. Gries and F. B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, Berlin, 1993.
- [11] David Gries. *The Science of Programming*. Springer, New York, 1981.
- [12] E.C.R. Hehner. What’s wrong with formal programming methods? In *Advances in Computing and Information*, volume 497 of *Lecture Notes in Computer Science*, pages 2–23. Springer, Berlin, 1991. <http://www.cs.toronto.edu/~hehner/publist.html>.
- [13] E.C.R. Hehner. Specifications, Programs, and Total Correctness. *Science of Computer Programming*, 34:191–205, 1999. <http://www.cs.toronto.edu/~hehner/publist.html>.
- [14] E.C.R. Hehner. Variables and Scopes Considered Formally. *Information Processing Letters*, 79:33–38, 2001. <http://www.cs.toronto.edu/~hehner/publist.html>.
- [15] E.C.R. Hehner. How to solve math problems., 2004. <http://www.cs.toronto.edu/~hehner/publist.html>.
- [16] E.C.R. Hehner. A Practical Theory of Programming, 2004. <http://www.cs.toronto.edu/~hehner/aPToP/>.

- [17] Maritta Heisel. Anforderungen und Spezifikationen, Vorlesungsscript, TU Ilmenau, 2002. <http://www-ia.tu-ilmenau.de/IPI/SWT/skripte/anfspez.pdf>.
- [18] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.
- [19] C. A. R. Hoare. Proof of Correctness of Data Representation. *Acta Informatica*, 1:271–281, 1972.
- [20] C.A.R. Hoare. An Overview of some Formal Methods for Program Design. *IEEE Computer*, 20(9):85–91, September 1987.
- [21] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall series in Computer Science, 1998.
- [22] Michael Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison Wesley, 2000.
- [23] Michael Jackson. The Real World. In *Proceedings of a Symposium to celebrate the work of C A R Hoare*, 2000.
- [24] Micheal Jackson and Pamela Zave. "domain descriptions". In *Proc. RE'93 - 1st Intl. IEEE Symp. on Requirements Engineering*, pages 56–64, jan 1993.
- [25] Hrsg. Johannes Siedersleben. Quasar - die standardarchitektur von sd&m. Technical report, software, design & management AG, 2002. <http://www.sdm.de/de/unternehmen/fe/quasar/index.html>.
- [26] Hrsg. Johannes Siedersleben. *Softwaretechnik - Praxiswissen für Softwareingenieure*. Carl Hanser Verlag, 2nd edition, 2003.
- [27] Donald E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, may 1984.
- [28] John Hughes Koen Claessen. QuickCheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Notices*, 35, September 2000.
- [29] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science, C.A.R. Hoare Series Editor. Prentice Hall International, Hemel Hempstead, UK, 1988.
- [30] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. International Series in Computer Science. Prentice Hall, New York, NY, 1990.
- [31] Bertrand Meyer. Static Typing and other mysteries of life. In *OOPSLA Conference*, 1995.
- [32] Bertrand Meyer. *Object-Oriented Software Construction*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, second edition, 1997.
- [33] Bertrand Meyer. The Next Software Breakthrough. *Computer*, 30(7):113–114, July 1997.
- [34] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2 edition, 1994. <http://users.comlab.ox.ac.uk/carroll.morgan/PfS/>.
- [35] S.L. Peyton-Jones and J.-M. Eber. Composing contracts: An adventure in financial engineering. In *Proc. ICFP*. ACM, 2000.
- [36] B. I. Witt R. C. Linger, H. D. Mills. *Structured Programming: Theory and Practice*. Addison Wesley, 1979.

- [37] Scott. Logic and Programming Languages (1976). In *ACM Turing Award Lectures: The First Twenty Years, ACM Press Anthology Series*. ACM Press, New York, Addison-Wesley, 1987.
- [38] E. Sekerinski and K. Sere, editors. *Program Development by Refinement : Case Studies Using the B Method*, chapter Foreword. Number ISBN 1-85233-053-8 in Formal Approaches to Computing and Information Technology. Springer Verlag London, December 1998.
- [39] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
- [40] Antero Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.
- [41] Robert Will. Abstract Algebraic Data Structures: a Specification and an Implementation. <http://www.stud.tu-ilmenau.de/~robertw/dessy/fun/>.
- [42] Robert Will. Explaining the Diplomacy Rules with Animation. in Vorbereitung.
- [43] Robert Will. Modularity without Abstraction is no more than Syntactic Sugar, december 2003. http://www.stud.tu-ilmenau.de/~robertw/eiffel/modularity_abstraction.txt.
- [44] Robert Will. Pragmatik von Typsystemen, Objektorientierung und Eiffel, Studienarbeit TU Ilmenau, october 2003. <http://www.stud.tu-ilmenau.de/~robertw/sa.pdf>.
- [45] Robert Will. Problem Diagrams and B, Praktikumsbericht, April 2003. <http://www.stud.tu-ilmenau.de/~robertw/rapport.pdf>.
- [46] Robert Will. Resizable Arrays in Dessy, August 2003. <http://matrix.fem.tu-ilmenau.de/~robertw/resizables>.
- [47] Robert Will. Ausnahmen mit Methode, february 2004. http://www.stud.tu-ilmenau.de/~robertw/eiffel/siedi_exceptions.txt.
- [48] Robert Will. Disciplined Exceptions -or- Error Management is Risk Management, february 2004. <http://www.stud.tu-ilmenau.de/~robertw/eiffel/cov/b.txt>.
- [49] Robert Will. Formalisations of the Diplomacy Rules, 2004. <http://www.stud.tu-ilmenau.de/~robertw/dessy/fun/examples/diplomacy>.
- [50] Robert Will. How C++ succeeded in failing., july 2004. <http://www.stud.tu-ilmenau.de/~robertw/cpp.htm>.
- [51] Niklaus Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, April 1971.
- [52] Pamela Zave and Michael Jackson. Conjunction as Composition. *acm Transactions of Software Engineering and Methodology*, 2(4):379–411, October 1993.
- [53] Pamela Zave and Michael Jackson. Where Do Operations Come From?: A Multiparadigm Specification Technique. *IEEE Transactions on Software Engineering*, 22(7):508–528, July 1996.
- [54] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.

Lese-Tipp: Reißen Sie das Literaturverzeichnis heraus und verwenden es als Lesezeichen. So haben Sie die Referenzen immer zur Hand!

A Thesen

1. Bei der Programmierung geht es manchmal um das Formalisieren von Anforderungen in einer Programmiersprache und manchmal um das Herleiten einer effizienteren Implementierung aus einer eindeutigen, aber einfacheren Spezifikation. Meistens geht es aber um beides gleichzeitig: Formalisieren und Operationalisieren!
2. Für jeden der beiden Teilbereiche gelten ganz unterschiedliche Methoden und daher ist es die erste Aufgabe des Programmiers beide zu trennen. Selbst-spezifizierender Code (d.h. formalisierte Anforderungen) soll als solcher gekennzeichnet werden, und der ganze Rest soll eine separate, einfache und präzise Spezifikation bekommen.
3. Ob ein Programmteil eine formale Spezifikation hat oder nicht — Tests braucht er in jedem Fall.
4. Alle äußeren Qualitätsmerkmale von Software außer der Effizienz gehen in die Spezifikation ein, die Implementierung misst sich nur an der Spezifikation und inneren Qualitätsmerkmalen.
5. Von allen formalen Theorien entspricht die *Verfeinerung* am besten den modernen Programmiermethoden. Programme und Spezifikationen sind Bool'sche Ausdrücke, die Berechnungen beschreiben. Programme unterscheiden sich prinzipiell nur durch ihre Effizienz von Spezifikationen.
6. Programmentwurf besteht darin, jeden Programmteil so einfach wie möglich zu gestalten. Einfachere Implementierungen sind leichter zu erstellen, zu warten und sie bieten auch Fehlern weniger Platz und machen Tests effektiver.
7. Wenn man Schnittstellen explizit dokumentiert, neigt man dazu sie besser zu entwerfen, und sie werden tendenziell einfacher.
8. Ein Software-Entwurf kann nur in Bezug auf eine konkrete Spezifikation und im Vergleich mit Alternativen bewertet werden.
9. Abstrakte Datentypen unterscheiden wir in unveränderliche (Elemente einer Algebra, die durch Funktionen ineinander umgewandelt werden) und veränderliche (kleine Maschinen, deren Zustand durch Befehle verändert wird). Die unveränderlichen Objekte sind prinzipiell einfacher zu benutzen, alle Methoden der funktionalen Programmierung können auf sie angewendet werden. Die veränderlichen Objekte mit ihren Referenzen untereinander machen das Gesamt-Datenmodell eines Programms aus.
10. Vererbung dient dazu gemeinsame Spezifikationen und Implementierungen auszufaktorie- ren. Über externe Vererbung kann ein Kunde mehrere Zulieferer transparent über die gleiche Schnittstelle benutzen. Über interne Vererbung kann sich eine Klasse zusätzliche Funktionalität holen, ohne dass ihre Kunden davon betroffen sind.

Diese Thesen enthalten sowohl alte als auch neue Erkenntnisse. Eine Aufzählung der spezifischen Beiträge dieser Arbeit findet sich in Abschnitt 8.1.